DOCTORAT BRETAGNE LOIRE / MATHSTIC

Nantes Université

THÈSE DE DOCTORAT DE

NANTES UNIVERSITÉ

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

Par
Meven LENNON-BERTRAND

Bidirectional Typing for the Calculus of Inductive Constructions

Typage Bidirectionnel pour le Calcul des Constructions Inductives

Thèse présentée et soutenue à Nantes, le 24 Juin 2022
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes

Rapporteurs avant soutenance :

Neel KRISHNASWAMI	Associate Professor, University of Cambridge
Conor MCBRIDE	Reader, University of Strathclyde

Composition du Jury :

Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition du jury doit être revue pour s'assurer qu'elle est conforme et devra être répercutée sur la couverture de thèse

Président :	Prénom NOM	Fonction et établissement d'exercice *(à préciser après la soutenance)*
Examinateurs :	Jesper COCKX	Assistant Professor, TU Delft
	Herman GEUVERS	Professor, Radboud Universiteit Nijmegen
	Hugo HERBELIN	Directeur de Recherche, Inria Paris
	Assia MAHBOUBI	Directrice de Recherche, Inria Rennes
	Christine PAULIN-MOHRING	Professeure des Universités, Université Paris Sud
Directeur de thèse :	Nicolas TABAREAU	Directeur de Recherche, Inria Rennes
Membre invité :	Matthieu SOZEAU	Chargé de Recherche, Inria Rennes

Bidirectional Typing in the Calculus of Inductive Constructions

Doctoral Thesis

Meven Lennon-Bertrand

June 17, 2022

À Tangi,
Parce que je sais que tu aurais été le premier à te réjouir avec moi de là où j'en suis arrivé… et le premier à me taquiner pour me garder les pieds sur terre.

Abstract

This thesis broadly considers the question of giving a bidirectional treatment of the Calculus of Constructions (CIC), which underpins the proof assistant Coq, under three different angles corresponding to its three parts.

It first considers the question of giving a bidirectional account of CIC from a theoretical point of view. It contains the exposition of such a bidirectional presentation of CIC, with the general discipline that led to it. Follow a proof of equivalence between this presentation and the standard one. This equivalence is then used to establish properties of CIC that are hard to obtain in the standard setting: existence of principal types, and strengthening.

The second part sets on to formalize the idea of the first one, in the setting of the MetaCoq project, which aims at formalizing the meta-theory CIC in Coq, and to implement a kernel that is proven correct and complete. The formalized bidirectional structure supplies an intermediate between the high-level specification and the algorithm, which is key in order to prove that the kernel is complete.

Finally, the last part considers the question of designing an extension of CIC incorporating ideas from gradual typing, with the aim of bringing more flexibility to development in Coq. The bidirectional structure is once again valuable, as the characteristics of gradual typing – in particular the way it relaxes conversion – make it impossible to base the extension on the standard presentation of CIC.

Acknowledgments

I wouldn't have got halfway to the end of this thesis without the many great people around me, so let me try and thank them here for all they have brought me.

Obviously, the one person this thesis owes the most to is my advisor Nicolas. I feel extremely lucky to have had the possibility to run free and do my things, while still knowing that he was in the next office with an open door whenever I got an issue. I am proud that I can call myself his student.

Beyond Nicolas, I am very honoured that my jury members accepted to be part of it. Neel and Conor, because of their deep knowledge about bidirectional typing and programming language theory in general. Herman, because my year in Nijmegen has been an important one for where I am today. Christine and Hugo, because they shaped the great tool that Coq is today; and, for Hugo, because he was the one who introduced me both to Coq and to research. Jesper, because I have a great respect his work on Agda. I hope there are still many pages to write in the collaboration between our two close but yet subtly different worlds. Matthieu, because his work on MetaCoq is impressive, and he still is kind enough to let me leave him discharge on him my ugliest lemmas. Finally, Assia, because I admire both her achievements and her genuine humanity.

During these three and a half years, I have learned so much from the researchers around me, and I am very grateful for that. From Pierre-Marie, that you should not be afraid to stand by your ideas. From Assia, that this does not mean you have to crush others, but that you can instead learn a great deal from them, exactly because they took another path. From Guillaume, that there are some non-negotiable values in academia that you cannot just ignore. From Guilhem, I hopefully learned a bit about teaching. Thanks for helping me navigate the meanders of the university and for trusting me with your exercise sessions. From Matthieu, I learned most of what I know about another kind of meanders, those of Coq and formalization. Éric taught me that having a solid technique is only useful if readers get that far in your paper, and showed me how to achieve that.

But Gallinette is not only permanent researchers, and I learned an equally great deal from my *senpais*. Marie and Étienne came up with the brilliant idea of the Quésaco seminars. Yannick still patiently answer my random questions, even after so many of them. Théo guided me through the intricacies of the PhD, which is no small feat. Loïc does not really fit in the *senpai* category, after all of the two thesis siblings I am the one coming out first. Still, I learned a lot with him, from precious insight on normalization to painting beautiful banners. Good luck with your own writing, and see you on the other side! Last but not least, Kenji taught me just too many things to be all recorded here. I'll miss our random but always worthwhile exchanges between computer screens and your valuable ideas on dependent types and vegetables alike.

Learning is nice, but it's not everything. Happily, the Gallinette members are not just cool researchers, they are also wonderful people, and it's been

an immense pleasure to work in that team. Thanks to all those I have already mentioned and to Pierre (Vial), Maxime, Simon, Matthieu (Piquerez), Ambroise, Xavier, Martin, Enzo, Pierre (Benjamin? Giraud? How do I cite you?), Hamza, Chris, Gaëtan for the seminars, coffee discussion, corridor talks, online chats, beer disputes, and all the other conversations. A special acknowledgment to Ambroise for thanking me in advance in his thesis, the prophecy has been accomplished. Following this now established tradition, I hereby thank in advance Martin, Enzo, Pierre, Hamza and Chris for thanking me in their own theses when the time comes. Best of luck to get there. Another special thanks to those who proofread this thesis even with the rushy calendar, and especially to Martin who found the courage to go through all of it. You made it much better than it was. And let me not forget Anne-Claire, who kindly watches over our unruly lot.

Au-delà de Gallinette, je me dois aussi de mentionner les doctorant·e·s du LS2N. Entre le Covid et la rédaction, je n'ai que trop peu profité d'eux, mais quelle chouette équipe ! Plus loin de Nantes, je n'oublie pas mon camarade Curry-Howardien Rémy. Un jour ce livre de théorie des catégories verra le jour, j'en suis sûr. Merci aussi à toute l'équipe des CHATS, Sylvain, Bertrand, Rémi, Brigitte, Audrey, Marianne, Baptiste, Mélanie, et tou·te·s les profs et élèves du lycée Michelet. Ces trois années d'art et de sciences n'ont pas exactement été de tout repos, mais j'ai beaucoup de plaisir à partager un peu de mon enthousiasme pour les mathématiques et l'informatique, et de tant recevoir en retour. Enfin, j'ai été très heureux de faire partie de l'éphémère collectif des précaires à l'hiver 2020, hélas presque aussi vite dispersé qu'il s'est formé. Merci à eux et à tous les autres qui luttent pour faire de l'université et du monde des endroits un peu meilleurs.

These pandemics years have not exactly been the best for exchanges. Yet, I feel very fortunate to be part of the thriving proof assistants and types communities. Thanks for all the online questions, answers and discussions, for giving nice talks and listening to mine. Hope to finally meet you all in person soon. And thanks to Andrej for giving me the occasion to have the first (!) scientific trip of my PhD in Ljubljana.

Je n'aurais jamais pu arriver là où j'en suis sans les nombreux·ses enseignant·e·s qui ont croisé ma longue route de la maternelle au master. Ils et elles ont su nourrir ma curiosité, et me montrer que ce n'est pas parce qu'un sujet est difficile que son apprentissage doit être dur. Je ne saurais toutes et tous les nommer, mais je veux en particulier remercier Jean-Michel Rey et François Sauvageot pour m'avoir fait entrer dans le monde des grandes personnes en posant les fondations sur lesquelles tout le reste s'est construit, et Daniel Hirschkoff pour avoir veillé avec bienveillance sur mes années à l'ENS et m'avoir un jour parlé de cette petite équipe sympathique qui fait du Coq à Nantes. I have already mentioned Herman, but let me extend the praise to Freek, Jurriaan, and my other Dutch professors, for having been great teachers and having introduced me to proof assistants, type theory, and many of the ideas I still use today.

Si on en croit von Neumann, si les gens ne croient pas que les mathématiques sont simples, c'est qu'ils ne réalisent pas à quel point la vie est compliquée. Fort heureusement, je suis au moins aussi bien entouré dans la vie qu'en mathématiques.

Bien sûr, par ma famille. Merci évidemment à mes parents d'avoir été là pour moi depuis toujours, de m'avoir soutenu de toutes les manières pos-

sibles, dans tous mes choix. Et d'avoir été d'attentifs et exigeants lecteurs et relecteurs de cette introduction. Merci à Tangi pour les leçons si profondes et les moments si légers. Et pour être toujours avec moi. Merci à Malo et Maïan d'avoir repris avec brio le flambeau de la taquinerie de ses mains, et à Marc et Karine de nous avoir fait grandir, eux et moi. Enfin, merci à Papi de toujours croire en ma réussite avec une confiance sans faille, même sans y comprendre grand-chose.

Je serais devenu fou si je n'avais pas pu sortir de mon bureau pour prendre un peu de grand air, merci donc à Karlijn (not sure our trips qualify as simply "a little fresh air" though), Olivier (pour le trad et pour les olives), Daniel (à quand la grande voie en 7 ?), Manon (les force 5, quelle aventure), Jodie, Émile, Alice, et Sacha. J'accepterai un jour qu'avec les Zouzous on passe plus de temps de part et d'autre d'une table bien garnie que d'une corde à double. Merci à Léo pour avoir été non seulement un excellent compagnon de cordée, mais aussi un compagnon de mathématiques, de musique, bref de vie ! Et à Paul de compléter avec brio le trio et pour sa curiosité scientifique toujours passionnante. Comment oublier les lyonnais qui ont toujours eu une place pour moi sur un canapé quand je les rejoignais pour une soirée de fête ou à jouer (Dune c'est le feu). Merci donc à Hugo, Angèle, Solène, Audrey, Clément, Loïs, Colin et tous les autres. Enfin, comment parler de soirées sans évoquer celles à la Milleterie, organisées et surtout illustrées avec un talent rare par Valentin.

L'évasion n'est pas tout, j'ai aussi été entouré par une équipe de choc dans ma vie nantaise. Allan et Maxou sont aussi généreux comme hôtes que féroces aux aventuriers du rail, et c'est parfait ainsi. Marine, qui même si elle réussit l'exploit d'être encore plus busy que moi reste une ancre à laquelle je sais que je peux me raccrocher. Mes colocataires, Julien, Laurine et Martin, pour avoir partagé ma vie et mon quotidien, les rires et les moments moins funs, et avoir survécu aux confinements ensemble sans qu'on s'entretue. Lucie, qui à ce stade rentre presque dans la catégorie précédente (la team Olivettes restera). Enfin merci évidemment à Jo, qui est à la fois une colocataire, une ancre, une féroce joueuse, une compagne d'aventure, et bien plus encore, pour m'avoir nourri pendant ma rédaction, traîné à la mer, ou pallié mon inculture cinématographique quand je n'en pouvais plus… Hâte de lire tes remerciements à toi, j'espère que j'y serai en aussi bonne place.

How to Read This Thesis

As I think screen reading will be the medium used by most of my readers, I use hyperlinking as much as possible inside the document. While it is not visible – in order to keep the text readable –, most technical keywords are actually linked to the place of their definitions. For instance, bidirectional typing links to the place in Chapter 2 where the notion is introduced. This definition itself is put into emphasis like the following *example*. I might cheat a bit and introduce a notion twice, once on a high level in an introductory section, and a second time precisely later on, in which case the link point to the precise definition. Most notations are also linked: if you wonder what the symbol $\sqsubseteq^{\mathrm{ob}}$ means again, just click on it!

The main text has large margins, which I use and abuse for notes, small figures and references. Hopefully this reduces the need to go back and forth between the main text and information too far away. Regarding figures, rather than having large, bulky ones, I tried to keep them as close as possible to their explanation. This means that they are often split in multiple small fragments, so that each part of the figure goes with its explanation. In such cases, the fragments should really be understood as different parts of one and the same figure. To indicate this, the fragments share the same figure number, such as Figures 3.1a to 3.2e which define a single system, one rule at a time.

Finally, although I primarily intend this document to be read on screen, I tried to keep it adapted for printing. In particular, no information should be conveyed using only colour, though I use it to ease readability.

Contents

Abstract v

Acknowledgments vi

How to Read This Thesis ix

Contents xi

1. Résumé en français 1
- 1.1. Une très courte histoire de la logique . . . 2
- 1.2. Les ordinateurs entrent en scène . . . 5
- 1.3. Coq et son noyau . . . 7
- 1.4. Et cette thèse, alors ? . . . 10

2. Introduction 13
- 2.1. A Very Short History of Logic . . . 14
- 2.2. Computers Enter the Scene . . . 16
- 2.3. Coq and Its Kernel . . . 19
- 2.4. And this Thesis ? . . . 21

3. The Calculus of Inductive Constructions 23
- 3.1. Terms and Types . . . 23
- 3.2. Functional Core: CC_ω . . . 24
- 3.3. 50 Shades of Conversion . . . 26
- 3.4. The Good Properties . . . 30
- 3.5. Adding Inductive Types: CIC . . . 35
- 3.6. Beyond CIC: PCUIC . . . 41

Bidirectional Calculus of Inductive Constructions 47

4. Warm-up: CCω 51
- 4.1. Turning CC_ω Bidirectional . . . 51
- 4.2. Properties of the Bidirectional System . . . 55

5. Bidirectional PCUIC 61
- 5.1. Cumulativity . . . 61
- 5.2. Inductive Types . . . 62

6. Bidirectional Conversion 65
- 6.1. Bidirectional Conversion . . . 66
- 6.2. Untyped Presentation . . . 69
- 6.3. McBride's Discipline . . . 70
- 6.4. Equivalence of the presentations . . . 74

A Certified Kernel for Coq, in Coq **79**

7. Formalized Meta-Theory of PCUIC **83**
7.1. Setting up the Definitions: Terms, Cumulativity and Types . . . 83
7.2. Stabilities . . . 90
7.3. Confluence . . . 91
7.4. The Road to Subject Reduction . . . 93
7.5. Normalization . . . 95

8. Building a Certified Kernel **97**
8.1. Formalized Bidirectional Typing . . . 97
8.2. Before Typing: Environment Querying and Cumulativity Checking . . . 99
8.3. Correct and Complete Inference . . . 101
8.4. Beyond Typing: Environment Checking and Re-Typing . . . 104

Bidirectional Elaboration for Gradual Typing **105**

9. Gradual Typing Meets Dependent Types **109**
9.1. Safety and Normalization, Endangered . . . 112
9.2. Non-Gradual Approaches . . . 113
9.3. Gradual Simple Types . . . 115
9.4. Graduality and Dependent Types . . . 119
9.5. The Fire Triangle of Graduality . . . 122
9.6. GCIC: An Overview . . . 124

10. From GCIC to CastCIC: Bidirectional Elaboration **129**
10.1. CastCIC . . . 129
10.2. Bidirectional Elaboration: from GCIC to CastCIC . . . 135
10.3. Precision is a Simulation for Reduction . . . 141
10.4. Properties of GCIC . . . 147

11. Beyond CastCIC: Models, Indices and Pure Reasoning **151**
11.1. Realizing CastCIC . . . 151
11.2. The issue with indices: gradual vectors and equalities . . . 153
11.3. A Reasonably Gradual Type Theory . . . 156

12. Perspectives **159**
12.1. Bidirectional Typing for Dependent Types . . . 159
12.2. MetaCoq's Future . . . 160
12.3. Gradual CIC . . . 161

Appendix **163**

A. Names for Type Systems **165**

Bibliography **167**

Résumé en français 1.

This chapter is an introduction intended for French-speaking readers. If your English is better than your French, you should instead read Chapter 2, its translation in English.

"Coq est un vieil homme maintenant, et il a de nombreuses cicatrices."

[Har20, citant Assia Mahboubi, traduction personnelle]

[Har20] : Hartnett (2020), *Building the Mathematical Library of the Future*

Cette thèse se situe dans le domaine de la théorie des types,[1] lui-même au croisement entre informatique et logique mathématique. Un de ses objectifs est de donner des fondements théoriques et pratiques à des outils informatiques assistant les humains dans la construction et la vérification de preuves – au sens mathématique du terme. De tels outils sont appelés assistants à la preuve, et, dans cette thèse, il sera en particulier beaucoup question de l'un d'entre eux, sur lequel mon travail s'est principalement concentré : Coq.

1 : Si vous ne connaissez pas la signification de ce terme, ou d'un autre qui apparait dans cette introduction, continuez votre lecture ! Ils seront expliqués en temps voulu.

1.1 Une très courte histoire de la logique 2
1.1.1 Les syllogismes 2
1.1.2 Les débuts de la logique mathématique : vers un fondement formel 3
1.1.3 La crise des fondements 3
1.1.4 L'incomplétude 4
1.1.5 Une situation satisfaisante ? . 4
1.2 Les ordinateurs entrent en scène 5
1.2.1 Les assistants à la preuve . . . 5
1.2.2 Logique, programmation et théorie des types 6
1.3 Coq et son noyau 7
1.3.1 Le noyau 7
1.3.2 MetaCoq 8
1.3.3 Typage bidirectionnel 9
1.3.4 Types graduels 9
1.4 Et cette thèse, alors ? 10
1.4.1 Théorie du typage bidirectionnel 10
1.4.2 Typage bidirectionnel dans MetaCoq 10
1.4.3 Élaboration bidirectionnelle pour le typage graduel 10
1.4.4 Contributions techniques et publications 11

Durant leurs plus de 50 ans d'existence, les assistants à la preuve sont devenus une technologie bien établie. Avec l'évolution du domaine, les outils sont devenus de plus en plus complexes, ce qui les rend à la fois de plus en plus puissants, mais aussi de plus en plus susceptibles de contenir des bugs critiques, cachés dans des recoins obscurs. Alors que les assistants à la preuve sont graduellement adoptés dans un nombre grandissant de communautés attachées à un haut niveau de fiabilité, cette situation n'est pas tenable. La solution historique consistant à placer sa confiance dans un petit noyau fiable – dénommée critère de De Bruijn –, n'est tout simplement pas suffisante si l'on veut avancer en intégrant de nouvelles fonctionnalités pour suivre les besoins des utilisateurs et utilisatrices.

Il y a une solution simple à ce problème : les assistants à la preuve sont utilisés depuis des décennies pour certifier la correction de programmes. Pourquoi ne pourraient-ils pas prouver *leur propre correction* ? Après tout, s'il s'agit là du critère le plus restrictif pour mesurer la confiance qu'on peut accorder à un logiciel, il devrait s'appliquer en premier lieu aux logiciels utilisés pour justifier cette confiance. Pour l'assistant à la preuve Coq, cette ambition est portée par le projet MetaCoq, qui vise à construire un nouveau noyau pour Coq qui soit entièrement prouvé correct. À terme, l'objectif est de pouvoir tout simplement remplacer le noyau actuel, et on doit donc prendre en compte toute sa complexité.

Afin de pouvoir atteindre ce but, il est nécessaire d'étudier plus en profondeur les structures à l'œuvre dans le noyau. En particulier, son algorithme de typage est *bidirectionnel*, ce qui signifie qu'il alterne en permanence entre la résolution de deux problèmes proches, mais distincts : l'*inférence* – trouver un type pour un terme – et la *vérification* – vérifier qu'un type donné convient pour un terme. Bien que cette structure soit cruciale pour relier la spécification du système de type à son implémentation, elle a été relativement peu étudiée dans le contexte du Calcul des Constructions Inductives (CIC), le fondement théorique de Coq – mais aussi de ses cousins Lean, Agda…

Cette thèse vise à remplir ce vide, en fournissant une étude rigoureuse d'un CIC bidirectionnel, formalisée dans le cadre offert par le projet MetaCoq. Celle-ci est un ingrédient clé dans la première preuve de correction et de complétude d'un algorithme de typage pour un noyau réaliste d'assistant à la preuve. Elle a également permis de découvrir des bugs dans le noyau de Coq qui étaient jusque-là passés inaperçus.

Mais le typage bidirectionnel est également un outil théorique intéressant en lui-même, donnant un contrôle précieux sur le calcul. En particulier, c'est une pièce nécessaire dans la conception d'une extension graduelle à CIC, GCIC. Le typage graduel vise à apporter aux programmeurs et programmeuses à la fois la flexibilité de développement offerte par le typage dynamique, et les garanties fortes données par le typage statique, dans un seul et même langage. GCIC cherche à fournir cette flexibilité aux types dépendants, et, en utilisant la puissance de la correspondance de Curry-Howard, à l'écriture de preuve. Mais cette entreprise bute sur des difficultés que seul le cadre bidirectionnel permet de résoudre.

Pour replacer ce travail dans son contexte large, le reste de cette introduction commence par une très courte histoire de la logique mathématique (Section 1.1), qui expose les principales problématiques de ce domaine. Suit une présentation des liens entre logique et informatique, par l'intermédiaire des assistants à la preuve (Section 1.2). La Section 1.3 s'intéresse plus particulièrement aux questions de recherche sur lesquelles j'ai travaillé : typage bidirectionnel, MetaCoq et typage graduel. Enfin la Section 1.4 résume mes contributions par cette thèse.

1.1. Une très courte histoire de la logique

1.1.1. Les syllogismes

La question principale à laquelle la logique cherche à répondre est celle de trouver des critères afin de déterminer si un raisonnement est valide. Dans la tradition occidentale, on peut faire remonter l'étude de cette problématique à l'Antiquité, et notamment à Aristote, avec son *Organon*. L'apport majeur de ce travail est d'introduire la notion de syllogisme. Il s'agit de fragments simples de raisonnement, dont la validité tient au fait qu'ils suivent une structure fixée, et non à un contenu particulier.[2] Si un raisonnement complexe est construit en assemblant ces syllogismes, celui-ci pris dans son entier doit nécessairement être valide, puisque chacun des fragments assemblés l'est. Il y a ici deux idées importantes.

2 : Le plus connu est probablement le syllogisme *Barbara*, dont un exemple est : *tous les humains sont mortels ; Socrate est humain ; donc Socrate est mortel.*

La première est qu'un raisonnement peut être valide ou non du simple fait de sa structure, indépendamment de son contenu. Il peut s'agir de syllogismes, mais aussi de bien d'autres systèmes. On en rencontrera un certain nombre au cours cette thèse !

La seconde est celle de la construction à partir de composantes élémentaires. En partant d'un système de règles de base qu'on a identifiées comme valides *a priori*, on a un moyen de s'assurer de la validité de raisonnements potentiellement très complexes. Il suffit pour cela de vérifier que ceux-ci peuvent être décomposés en un assemblage des composantes de base.

Pour les philosophes grecs, la logique est également pensée comme un outil de communication. Il s'agit de vérifier la validité de son propre raisonnement, mais surtout de se donner le moyen d'échanger celui-ci, en s'accordant sur un système logique formel.[3] Une personne voulant que ses conclusions soient acceptées par d'autres n'a plus qu'à exprimer son raisonnement de manière parfaitement précise dans le cadre d'un tel système.

3 : Les règles structurelles à respecter, comme celles des syllogismes.

À partir de cette époque, la logique en tant que discipline se concentre sur l'étude de cette structure qui sous-tend le raisonnement. L'enjeu principal est donc de construire un système formel, adapté à un domaine de raisonnement précis. Dans le cadre qui nous intéresse, celui de la logique mathématique, cela permet de donner un sens précis à ce qui constitue une preuve mathématique valide.

1.1.2. Les débuts de la logique mathématique : vers un fondement formel

À la suite d'Aristote, les mathématiciens et mathématiciennes se sont donc emparés de la logique, à la recherche d'un système formel pouvant servir de fondement rigoureux aux mathématiques. Les liens entre logique et mathématiques remontent à l'Antiquité grecque, mais la logique mathématique en tant que discipline indépendante s'est réellement établie durant le 19^{e} siècle, grâce à d'importants progrès sur deux aspects principaux.

Le premier a consisté à se dégager du langage dit naturel[4], inadapté à une description formellement précise de la déduction, et à concevoir à la place une nouvelle forme de langage spécifique à même servir de base au raisonnement mathématique. Une étape importante ici est le *Begriffsschift* de Frege [Fre79], qui, le premier, donne un langage formel suffisamment riche pour exprimer les mathématiques de manière satisfaisante. Son addition majeure est l'introduction de la notion de quantificateur, essentielle au langage mathématique, car ils permettent de fidèlement rendre compte des propriétés universelles[5] et existentielles[6].

4 : Par opposition aux langages formels qui apparaissent en mathématiques, informatique, etc.

[Fre79] : Frege (1879), *Begriffsschrift : Eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens*

5 : Par exemple : « Tout entier pair est la somme de deux nombres premiers ».

6 : Par exemple : « Il existe un réel dont le carré vaut 2 ».

Le second aspect a eu pour but de montrer que les mathématiques dans leur entier pouvaient être reconstruites à partir d'un petit nombre de propriétés simples. Une étape importante est la réduction de l'analyse aux propriétés des nombres réels, puis les constructions de ceux-ci à partir de l'arithmétique, données quasiment simultanément par entre autres Dedekind [Ded72] et Cantor [Can72] en 1872. De son côté, Peano [Pea89] propose une axiomatisation des nombres entiers proche de celle encore utilisée aujourd'hui. Enfin, Cantor à nouveau introduit la théorie des ensembles [Can83] comme un formalisme permettant de décrire tous les objets mathématiques sous la forme d'ensembles d'éléments.

[Ded72] : Dedekind (1872), *Stetigkeit und Undirrationale Zahlen*

[Can72] : Cantor (1872), *Ueber die Ausdehnung eines Satzes aus der Theorie der trigonometrischen Reihen*

[Pea89] : Peano (1889), *Arithmetices principia : Nova methodo exposita*

[Can83] : Cantor (1883), *Grundlagen einer allgemeinen Mannigfaltigkeitslehre. Ein mathematisch-philosophischer Versuch in der Lehre des Unendlichen*

1.1.3. La crise des fondements

Hélas, le système proposé dans le *Begriffsschift* est incohérent ! C'est-à-dire qu'il permet de prouver le faux, faisant s'écrouler le système logique.[7] Ce constat, fait par Russell en 1902[8] ouvre une période de crise, en remettant en doute les systèmes qui avaient commencé à s'imposer comme de bons candidats pour servir de fondements aux mathématiques – celui de Frege, mais surtout ceux de Cantor, affectés par les mêmes difficultés.

7 : Dans un système où le faux est prouvable, toutes les propositions le sont, ce qui est connu sous le nom de principe d'explosion. Un tel système où tout – et son contraire ! – est prouvable ne peut évidemment pas servir de fondement satisfaisant aux mathématiques.

8 : Dans une lettre à Frege dont ce dernier a rendu le contenu public dans Frege [Fre03, Nachwort, p. 253].

[Fre03] : Frege (1903), *Grundgesetze der Arithmetik*

Une possible solution est avancée dix ans plus tard par Russell et Whitehead dans leur *Principia Mathematica* [WR13], un énorme travail qui, non seulement, propose un système formel qui évite l'incohérence du *Begriffsschift*, mais qui, de plus, construit dans ce système une quantité importante de mathématiques, en particulier, une construction des entiers, de l'arithmétique et finalement des nombres réels.

[WR13] : Whitehead et al. (1913), *Principia Mathematica*

En parallèle, dans la continuité des travaux de Cantor, Zermelo [Zer08] et d'autres travaillent à fournir une version de la théorie des ensembles de Cantor qui soit cohérente. Ceci aboutit à ce qu'on appelle actuellement la théorie des ensembles de Zermelo-Fraenkel – ZF, ou ZFC quand on y ajoute l'axiome du choix [Zer04][9] –, qui semble également à même de fournir une base solide pour fonder les mathématiques.

[Zer08] : Zermelo (1908), *Untersuchungen über die Grundlagen der Mengenlehre I*

[Zer04] : Zermelo (1904), *Beweis, daß jede Menge wohlgeordnet werden kann*

9 : Un axiome très utile dans de nombreuses branches des mathématiques, mais qui est souvent traité séparément, car il est à la fois moins crucial que les autres axiomes de ZF et à l'origine de résultats contre-intuitifs.

1.1.4. L'incomplétude

La recherche d'un système formel adéquat pour servir de fondement aux mathématiques se heurte cependant à une seconde difficulté majeure : le théorème d'incomplétude de Gödel [Göd31]. Celui-ci affirme que tout système formel dans lequel on peut construire des nombres entiers comme ceux de Peano – donc *a fortiori* tout système suffisamment riche pour fonder les mathématiques – ne peut pas démontrer sa propre cohérence.[10] De ce fait, il n'existe pas de système qui puisse servir de base aux mathématiques avec une certitude formelle sur son adéquation. En effet, puisqu'on ne peut pas prouver la cohérence du système dans lui-même, il pourrait finalement s'avérer incohérent, ruinant les efforts fournis – exactement ce qui est arrivé au *Begriffsschift* de Frege. Et si on utilise un second système pour démontrer la cohérence du premier, on n'a fait que déplacer le problème : c'est maintenant sur la cohérence de ce second système que l'on repose.

[Göd31] : Gödel (1931), *Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme. I*

10 : À moins qu'il ne soit incohérent, auquel cas il peut *tout* démontrer, par le principe d'explosion, dont sa cohérence… et son incohérence !

Une conséquence importante de ce théorème est qu'un système suffisamment riche pour fonder les mathématiques est nécessairement incomplet.[11] Ainsi, dans la suite, il ne sera jamais question de vérité dans un sens absolu – ce qui n'aurait de sens que dans un système complet où tout énoncé est vrai ou faux –, mais uniquement de prouvabilité *relativement à un système donné*.

11 : Cela signifie qu'il existe des énoncés indépendants, à savoir des assertions qui ne sont pas démontrables, et dont la négation ne l'est pas non plus. La cohérence du système considéré en est un exemple.

1.1.5. Une situation satisfaisante ?

Malgré les difficultés mises à jour au début du 20^{e} siècle, les recherches en logique mathématique ont abouti au milieu du siècle à une situation globalement assez satisfaisante. D'abord, ZFC fournit un système formel raisonnable sur lequel fonder les mathématiques. Ensuite, la communauté mathématique est globalement convaincue qu'il serait *théoriquement* possible de rédiger les mathématiques dans leur ensemble en utilisant celui-ci. Cela suffit amplement à la plupart de ses membres, même si rares sont ceux se risquent à véritablement tenter l'expérience, dans la veine des *Principia Mathematica*.

En *pratique*, les choses sont toutes autres. Le développement et la vérification humaine de mathématiques formalisées[12] semble à la fois impossible et inintéressant. D'un côté, cela demanderait un effort considérable, car de

12 : C'est-à-dire effectivement exprimées dans un système formel fixé.

telles mathématiques nécessitent un niveau de précision extrêmement élevée, tant de la part de l'autrice de la preuve formelle que de sa lectrice. Dans le même temps, cela ne permettrait pas de réduire de manière significative les risques d'erreurs. Il serait en effet humainement très difficile de vérifier qu'un raisonnement suit bien les règles du système : une minuscule erreur peut facilement se cacher au milieu de milliers de pages de raisonnement formel. Enfin, décrire les mathématiques de cette façon noierait les intuitions mathématiques importantes, rendant la communication stérile.

Si on veut rendre les mathématiques formelles praticables et bénéficier des garanties qu'elles apportent en éliminant ces défauts rédhibitoires, il faut donc développer de nouveaux outils.

1.2. Les ordinateurs entrent en scène

Un nouvel élément vient cependant modifier radicalement cette situation : l'avènement des ordinateurs. En effet, l'informatique donne accès à de nouveaux outils, qui permettent de rendre à la fois possible et attrayante la formalisation des mathématiques.

1.2.1. Les assistants à la preuve

Les ordinateurs excellent là où les humains pèchent : leur spécialité est de traiter d'immenses volumes d'information de façon très précise, exactement le type de besoins que soulève la manipulation de mathématiques formalisées. C'est pourquoi dès le début des années 70[13] commencent à apparaître des outils informatiques servant à écrire et vérifier ces preuves formelles, que l'on appelle collectivement des *assistants à la preuve*. Via la formalisation des preuves et la vérification par l'ordinateur qu'elles suivent bien les règles du système logique sous-jacent, les assistants à la preuve donnent accès à une fiabilité bien plus élevée que celle permise par les preuves "informelles". Des mathématiciens reconnus, comme Voevodsky [Voe10], Hales [Hal12, Preface, p. xi], ou Scholze [Sch21] se sont d'ailleurs déjà emparés de cette technologie, en particulier dans le but de lever les incertitudes quant à la solidité de leur propre travail.

13 : Avec des systèmes comme Automath [dBru70] ou Mizar [Rud92].

[dBru70] : de Bruijn (1970), *The mathematical language AUTOMATH, its usage, and some of its extensions*

[Rud92] : Rudnicki (1992), *An overview of the Mizar project*

[Voe10] : Voevodsky (2010), *Univalent foundations project*

[Hal12] : Hales (2012), *Dense Sphere Packings : A Blueprint for Formal Proofs*

[Sch21] : Scholze (2021), *Half a year of the Liquid Tensor Experiment : Amazing developments*

De plus, le terme d'*assistant* à la preuve n'a pas été choisi au hasard : au-delà de la simple vérification, ils mettent à la disposition des utilisateurs et utilisatrices un large éventail d'outils pour faciliter la conception de preuves formelles. Ces outils permettent d'écrire les preuves à haut niveau et de manière interactive,[14] en laissant à l'assistant à la preuve le soin de construire les preuves formelles. Il peut s'agir de simples facilités comme la possibilité de visualiser la structure des preuves, de suivre l'utilisation des hypothèses, mais aussi de techniques beaucoup plus ambitieuses.

14 : Dans la plupart des assistants à la preuve modernes, la preuve finale est construite comme le résultat d'un échange entre la programmeuse et l'outil, plutôt qu'écrite d'un seul bloc.

En effet l'informatique rend possible l'automatisation de pans entiers de l'écriture de preuves, par exemple via l'utilisation d'un langage de tactiques [Del00], qui permet de programmer la génération de preuves. La construction automatique de preuve est par ailleurs un domaine de recherche à part entière, et la question de son intégration dans les assistants à la preuve y est un sujet actif [Bla+16; Eki+17]. L'informatique a également fait ses preuves dans le champ du calcul mathématique (calcul formel, analyse numérique),

[Del00] : Delahaye (2000), *A Tactic Language for the System Coq*

[Bla+16] : Blanchette et al. (2016), *Hammering towards QED*

[Eki+17] : Ekici et al. (2017), *SMTCoq : A plug-in for integrating SMT solvers into Coq*

et là encore des connexions prometteuses avec les assistants à la preuve commencent à voir le jour [LW22; MMS19].

[LW22] : Lewis et al. (2022), *A Bi-Directional Extensible Interface Between Lean and Mathematica*
[MMS19] : Mahboubi et al. (2019), *Formally Verified Approximations of Definite Integrals*

Enfin, si l'utilisation de l'informatique facilite l'écriture de preuve, les assistants à la preuve ouvrent inversement de nouvelles possibilités pour la programmation. Ils offrent en effet un cadre naturel dans lequel décrire au même endroit le code source d'un programme, sa spécification et la preuve formelle que cette dernière est remplie. On peut alors *prouver* que le programme s'exécute correctement, sans rencontrer de bug. Cette certitude mathématique est bien plus fiable que n'importe quelle batterie de tests ! Dans ce domaine, de nombreux projets ont déjà abouti à des programmes d'envergure, entièrement prouvés corrects : compilateur pour le langage C [Käs+17], implémentation du protocole Https [Bha+17], résolution d'équations différentielles [Imm18]...

[Käs+17] : Kästner et al. (2017), *Closing the Gap – The Formally Verified Optimizing Compiler CompCert*

[Bha+17] : Bhargavan et al. (2017), *Everest : Towards a Verified, Drop-in Replacement of HTTPS*

[Imm18] : Immler (2018), *A Verified ODE Solver and the Lorenz Attractor*

1.2.2. Logique, programmation et théorie des types

Pour fonctionner, les assistants à la preuve nécessitent comme fondement un système formel, correspondant aux "règles du jeu" mathématique qu'ils sont censés imposer. Ainsi, ils requièrent une étude renouvelée de la logique mathématique, mais dans le but pratique de construire des outils à la fois fonctionnels, puissants et faciles à utiliser. Il existe plusieurs familles d'assistants à la preuve, basées sur des systèmes formels relativement différents. Celle qui m'intéresse dans cette thèse est fondée sur la correspondance de Curry-Howard et la théorie des types dépendants. C'est à elle qu'appartient l'assistant à la preuve Coq [Coq22a] qui est au cœur de mon travail.

[Coq22a] : Coq Development Team (2022), *The Coq Proof Assistant*

Si on compare un programme informatique à un texte dans une langue naturelle, les *types* sont une sorte d'équivalent des catégories grammaticales. Cependant, contrairement aux langues naturelles, ces types sont conçus en même temps que le langage de programmation, de manière à refléter des propriétés des objets manipulés par celui-ci. Cela permet en premier lieu de détecter des erreurs manifestes. Par exemple, si une procédure attendant un objet de type "image" est appliquée à un objet de type "chaîne de caractères", une erreur pourra être rapportée à la programmeuse.[15] Mais les types sont très versatiles, et leur capacité à encoder des propriétés des programmes sous-jacents peut servir à la compilation, la documentation, et bien d'autres choses. Dans notre cadre, par exemple, les types correspondent à la validité d'un raisonnement logique.

15 : Un slogan dû à Milner [Mil78] affirme que « Les programmes bien typés ne peuvent pas mal s'exécuter. »

[Mil78] : Milner (1978), *A theory of type polymorphism in programming*

Cette idée est celle de la *correspondance de Curry-Howard*.[16] Plutôt qu'un théorème précis, il s'agit d'un concept très général, selon lequel il existe une ressemblance forte entre d'un côté le monde de la logique et des preuves, et de l'autre celui des programmes et de leurs types. On parle aussi d'ailleurs également de *correspondance preuves-programmes*.

16 : Explicitée la première fois dans des notes informelles de Howard datant de 1969 mais publiées seulement bien plus tard [How80], qui reprenaient des remarques antérieures de Curry [CFC58].

[How80] : Howard (1980), *The Formulae-as-Types Notion of Construction*

[CFC58] : Curry et al. (1958), *Combinatory Logic*

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

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

$$\frac{p : A \times B}{p.1 : A} \qquad \frac{p : A \times B}{p.2 : B}$$

Figure 1.1. Règles d'inférence pour la conjonction et de typage pour les paires

Un exemple valant mieux qu'un discours abstrait, on peut voir la correspondance à l'œuvre dans la Figure 1.1, sous la forme de règles d'inférence ou de typage : chaque bloc présente une règle, avec au-dessus de la barre les hypothèses, et en dessous la conclusion. Les trois premières règles gouvernent la conjonction logique "et", notée $\wedge$. La première signifie que pour déduire la proposition $A \wedge B$ ("A et B"), il suffit de déduire A et B individuellement. À l'inverse si on a comme hypothèse $A \wedge B$, alors on peut déduire à la fois A, et B. Les trois dernières règles gouvernent le type des paires $A \times B$. Une paire (a, b) construite à partir d'un premier objet a de type A[17]

17 : Ce qui est noté $a : A$.

et d'un second objet b de type B a le type $A \times B$. À l'inverse si p est de type $A \times B$, alors on peut en récupérer la première composante $p.1$, qui est de type A, et la seconde $p.2$, de type B. Si on efface les termes[18] des règles du bas, on obtient *exactement* les règles du haut ! Ainsi, le concept de paire en programmation correspond directement à celui de conjonction en logique : la preuve d'une conjonction est une paire de preuves.

18 : Dans ce contexte, on parle souvent de *termes* plutôt que de programmes, mais les deux sont synonymes.

Ceci s'étend bien au-delà du cas de la conjonction, à une correspondance générale entre d'une part les énoncés de la logique et leurs preuves, et d'autre part les types et les programmes. On peut voir les énoncés comme des types, et une preuve d'un énoncé comme un programme ayant le type qui lui correspond – ou l'inverse. Au-delà de la simple analogie entre formalismes d'origines différentes, cette correspondance est un outil puissant pour faire dialoguer deux mondes. En particulier, elle permet de relier deux problèmes *a priori* éloignés : vérifier qu'une preuve est correcte, et vérifier qu'un terme est bien typé. Dans les deux cas, il s'agit de vérifier qu'une construction – programme d'un côté, preuve de l'autre – respecte un ensemble de règles formelles garantissant qu'elle est bien formée.

La correspondance de Curry-Howard est donc idéale pour servir de fondements aux assistants à la preuve, puisqu'elle permet de voir un système formel comme une logique, tout en donnant accès à des idées venant de la large littérature sur les langages de programmation, notamment la théorie et l'implémentation des systèmes de types. Dans ce cadre, les *systèmes de types dépendants* forment une famille particulière de systèmes de types, dont la caractéristique principale est d'autoriser les types à dépendre de termes. L'exemple archétypique du point de vue de la programmation est le type $\mathbf{Ve}\,(A, n)$ des vecteurs de longueurs n, les listes contenant exactement n éléments de type A – avec n un entier. Ce type dépend de n, au sens où les habitants du type diffèrent suivant les valeurs de l'entier. Du point de vue de la logique, cette dépendance correspond à la quantification : si on veut exprimer une propriété universelle « pour tout x, on a $P(x)$ », on a besoin que la propriété P puisse dépendre de x. Grâce à cette capacité à exprimer la quantification, les types dépendants sont suffisamment riches et puissants pour servir de fondement aux mathématiques.

1.3. Coq et son noyau

Intéressons-nous maintenant un peu plus précisément à l'assistant à la preuve dont il sera le plus question dans cette thèse : Coq.

1.3.1. Le noyau, clé de voûte du système

Coq est basé sur la correspondance de Curry-Howard : les preuves sont vues comme des programmes dans un langage appelé Gallina, et leur vérification est effectuée par un algorithme proche de ceux utilisés pour les types des langages conventionnels. Cependant, si, dans les premières versions des années 80, les preuves Coq étaient écrite quasiment directement en Gallina, ce n'est actuellement plus du tout le cas. La raison est que la majeure partie de l'outil dans ses versions actuelles a pour but d'aider l'utilisatrice à générer une preuve correcte. C'est un véritable *assistant* à la

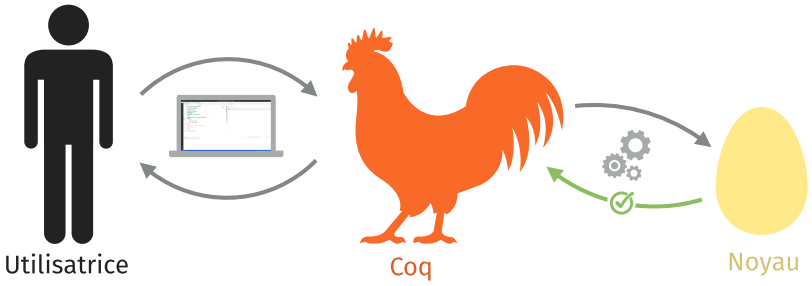

Figure 1.2. L'architecture schématique de Coq

preuve ! Ce fonctionnement est illustré ci-contre : l'utilisatrice échange interactivement avec Coq, qui utilise cette interaction pour générer un terme de preuve. Celui-ci est ensuite envoyé à une partie bien spécifique de l'outil, appelée *noyau*. C'est lui qui implémente l'algorithme de vérification de type, et s'assure ainsi de la correction des termes de preuve construits interactivement. Le noyau est donc l'élément crucial de Coq, car c'est lui – et lui seul – qui est responsable en dernier lieu de la validation des preuves. Cette architecture, qui isole clairement la partie critique du système, est appelée *critère de De Bruijn* [BG01] en hommage à l'un des pionniers des assistants à la preuve.

[BG01] : Barendregt et al. (2001), *Proof-Assistants Using Dependent Type Systems*

Si le reste de l'écosystème s'est beaucoup plus développé que le noyau depuis les débuts, celui-ci a cependant également évolué, en se complexifiant graduellement. Et comme tout développement logiciel, il n'est pas à l'abri de bugs[19]. Ceux-ci sont en général difficilement exploitables, encore plus sans s'en rendre compte. Néanmoins, ils existent, et le noyau tendant à toujours plus se complexifier ils risquent de continuer à apparaître.

19 : De l'ordre d'un bug détecté par an, une liste est maintenue à l'adresse suivante : https://github.com/coq/coq/blob/master/dev/doc/critical-bugs.

1.3.2. MetaCoq, une formalisation en Coq, pour Coq

Si on veut garantir un niveau de fiabilité le plus élevé possible, il faut donc de nouvelles idées. Le projet MetaCoq, a pour but de répondre à cette problématique. L'approche est simple : il s'agit d'utiliser Coq lui-même pour certifier la correction de son noyau.

Plus précisément, la première étape est de décrire le système de type sur lequel est basé le noyau, puis de démontrer ses propriétés théoriques. Il s'agit déjà d'une entreprise difficile : pour faciliter son utilisation, la théorie des types de Coq incorpore de nombreuses particularités complexes à traiter.

Une fois ces propriétés établies, la deuxième étape consiste à implémenter un algorithme de vérification de type ressemblant au maximum à celui du noyau, directement en Gallina.[20] On démontre en même temps qu'il est bien *correct*[21] et *complet*.[22]

Enfin, lors d'une troisième étape, on extrait de ce programme Gallina certifié un autre programme plus efficace, en effaçant le contenu lié à la preuve de correction pour ne garder que celui qui est algorithmiquement intéressant. Cette extraction est une étape complexe, mais cruciale si on veut remplacer le noyau actuel en conservant une efficacité raisonnable. C'est pourquoi on prouve là encore qu'elle est correcte[23], en la programmant à nouveau en Gallina.

20 : En effet, grâce à la correspondance de Curry-Howard, Gallina est certes un langage de preuve, mais aussi un véritable langage de programmation !

21 : Si l'algorithme prétend qu'un terme est bien typé, alors c'est bien le cas.

22 : L'algorithme répond bien affirmativement sur tous les termes bien typés.

23 : C'est-à-dire qu'elle préserve la sémantique des programmes.

1.3.3. Vérification, inférence et typage bidirectionnel

Afin de prouver que l'algorithme de typage de la deuxième étape est complet, il est très utile de passer par une spécification intermédiaire plus structurée que la description théorique de la première étape. En particulier, il est important de séparer deux questions proches, mais bien distinctes : d'une part, la vérification, où on cherche à *vérifier* qu'un terme a bien un type donné ; d'autre part, l'inférence, où on cherche à *trouver* un type pour un terme, s'il en existe un. L'algorithme de typage du noyau de Coq est *bidirectionnel*, c'est-à-dire qu'il alterne en permanence entre ces deux questions lorsqu'il vérifie qu'un terme est bien typé. Cette structure bidirectionnelle étant plus proche de l'algorithme, la décrire formellement mais séparément de l'implémentation permet de bien diviser les difficultés entre, d'un côté, son équivalence avec la présentation d'origine, et, de l'autre, la partie purement liée aux questions d'implémentation.

Dans le cas spécifique des types dépendants, bien que présent depuis l'origine dans les algorithmes de vérification de type [Hue89], le typage bidirectionnel a été relativement peu étudié pour lui-même. Pourtant, au-delà de son lien fort avec les algorithmes, cette approche présente également des avantages théoriques : elle permet, par sa structure plus contrainte que la présentation standard, d'obtenir des propriétés difficiles à démontrer dans ce cadre.

[Hue89] : Huet (1989), *The Constructive Engine*

1.3.4. Types graduels : un peu de flexibilité dans un monde désespérément statique

Il existe deux grandes approches de la vérification du type des programmes. Dans l'approche statique,[24] les types sont vérifiés en amont de l'exécution, alors que, dans l'approche dynamique, le bon typage des opérations est vérifié à la volée lors de cette même exécution. La discipline dynamique est plus flexible, parce qu'elle permet de vérifier exactement ce qui est nécessaire à la bonne exécution d'un programme. La rigidité du typage statique permet, elle, de détecter des erreurs plus tôt dans le développement, et impose des invariants utiles pour optimiser la compilation ou l'exécution.

24 : Qui est celle sur laquelle est basée Coq.

Plutôt que d'opter exclusivement pour l'une de ces deux approches, le *typage graduel* [Sie+15] vise à intégrer dans un même langage disciplines statiques et dynamiques. L'idée est d'avoir une première passe de vérification avant l'exécution, comme en typage statique, tout en laissant la possibilité de déférer une partie de la vérification à l'exécution, comme en typage dynamique. On a alors accès à tout un spectre d'options, d'une discipline totalement statique à une discipline totalement dynamique, en pouvant choisir finement quelles parties d'un programme on veut vérifier de quelle façon. En particulier, on peut faire évoluer la discipline au fur et à mesure d'un développement logiciel, pour bénéficier de la flexibilité du typage dynamique dans les phases précoces et des garanties du typage statique par la suite.

[Sie+15] : Siek et al. (2015), *Refined Criteria for Gradual Typing*

Comme le cas de MetaCoq l'illustre, Coq peut être utilisé comme un véritable langage de programmation. Mieux : son système de type permet d'exprimer des propriétés très complexes des programmes, et ainsi de vérifier avant même leur exécution que celles-ci sont bien respectées par le

code. Hélas, ces très fortes contraintes peuvent se retourner contre l'utilisatrice, en rendant plus difficile la phase de développement. En effet, il serait parfois bon de pouvoir lever temporairement les garanties très fortes du typage afin de faciliter l'expérimentation. Pour ce faire, on peut s'inspirer des idées du typage graduel, pour permettre un développement logiciel ou logique plus flexible. À nouveau, la correspondance de Curry-Howard est à l'œuvre, puisqu'on adapte des concepts venant du monde de la programmation au cadre de la logique.

1.4. Et cette thèse, alors ?

Mon travail de doctorat lui-même est centré principalement autour du typage bidirectionnel, sous trois aspects, correspondant aux trois parties de cette thèse. Elles sont précédées par le Chapitre 2, version anglophone de ce chapitre, et le Chapitre 3, qui introduit les principales notions techniques utilisées par la suite.

1.4.1. Théorie du typage bidirectionnel

La première partie (Bidirectional Calculus of Inductive Constructions) propose de combler une partie du manque théorique autour du typage bidirectionnel pour les types dépendants. Elle contient en particulier une preuve d'équivalence entre la présentation standard de la littérature et une présentation bidirectionnelle. Le Chapitre 4 présente les idées générales qui guident ce travail dans un cadre relativement simple, afin de faciliter leur exposition. Le Chapitre 5 montre comment étendre ces idées à un cadre plus réaliste, proche de la théorie des types implémentée en pratique dans Coq. Enfin le Chapitre 6 traite du statut particulier de la conversion[25] et des liens entre certains travaux récents sur ce sujet et le typage bidirectionnel.

25 : Cette notion cruciale permet d'intégrer dans la théorie des types dépendants l'idée de calcul des programmes.

1.4.2. Typage bidirectionnel dans MetaCoq

La seconde partie de cette thèse (A Certified Kernel for Coq, in Coq) s'intéresse au projet MetaCoq, et en particulier à la formalisation en Coq des idées présentées dans la première partie. Le Chapitre 7 donne une présentation générale du projet, tandis que le Chapitre 8 se concentre spécifiquement sur la preuve que le noyau implémenté par MetaCoq respecte sa spécification, et en particulier la preuve de complétude qui nécessite d'utiliser le typage bidirectionnel.

1.4.3. Élaboration bidirectionnelle pour le typage graduel

Enfin la troisième et dernière partie (Bidirectional Elaboration for Gradual Typing) présente mon travail dans le domaine des types graduels. Les types dépendants formant déjà des systèmes complexes, l'adaptation de ceux-ci à l'approche graduelle est particulièrement délicate. Un résumé des possibilités et difficultés est présenté en Chapitre 9. Un point intéressant à souligner est que la présentation habituelle des types dépendants s'avère

inadaptée, car trop flexible. Au contraire, la structure additionnelle apportée par le typage bidirectionnel permet de résoudre ces problèmes. Elle est de plus pertinente pour présenter l'élaboration de termes depuis un langage source dans un langage cible, une caractéristique importante des langages graduels. L'utilisation d'une élaboration bidirectionnelle, et les propriétés qu'elle permet d'obtenir, sont décrites en Chapitre 10. Enfin le Chapitre 11 décrit un travail dans la continuité de celui du Chapitre 10, mais qui n'est pas directement lié au typage bidirectionnel.

1.4.4. Contributions techniques et publications

Mon doctorat a débuté avec l'étude des types dépendants graduels. J'ai contribué avec Kenji Maillard, Nicolas Tabareau et Éric Tanter à Lennon-Bertrand et al. [Len+22], où nous étudions une extension graduelle pour le Calcul des Constructions Inductives. Ma contribution technique principale dans ce cadre correspond au Chapitre 10. L'étude fine de la littérature et le théorème d'impossibilité du Chapitre 9 auquel elle amène sont également tiré de cette publication. La seconde partie technique de Lennon-Bertrand et al. [Len+22], à laquelle j'ai participé mais dont l'auteur principal est Kenji Maillard, ainsi qu'un second article avec les mêmes auteurs et dans la continuité du précédent[26] sont présentés au Chapitre 11.

[Len+22] : Lennon-Bertrand et al. (2022), *Gradualizing the Calculus of Inductive Constructions*

26 : Maillard et al. [Mai+22], actuellement en phase de relecture.

[Mai+22] : Maillard et al. (2022), *A Reasonably Gradual Type Theory*

Ce travail ayant montré l'utilité d'un système de type bidirectionnel dépendant et le manque de résultats sur le sujet, j'ai choisi de l'étudier plus en détail, à la fois sur papier et par le biais d'une formalisation se basant sur MetaCoq. Ceci a donné lieu à une seconde publication [Len21], et correspond aux Chapitres 4 et 5 pour la partie théorique, ainsi qu'à la Section 8.1 pour la formalisation. Le bug de complétude du noyau de Coq découvert au cours de cette formalisation, ainsi que l'impact de cette découverte sur l'implémentation de Coq est présentée dans Sozeau, Lennon-Bertrand et Forster [SLF22].

[Len21] : Lennon-Bertrand (2021), *Complete Bidirectional Typing for the Calculus of Inductive Constructions*

[SLF22] : Sozeau et al. (2022), *The Curious Case of Case : Correct & Efficient Representation of Case Analysis in Coq and MetaCoq*

J'ai ensuite travaillé à l'intégration de cette formalisation à MetaCoq, et à son utilisation pour montrer la complétude du noyau qui y est implémenté.[27] Ceci correspond à la Section 8.3. Au-delà de cette contribution principale, j'ai également participé à ce projet sur d'autres points plus mineurs. Cette partie de mon travail de thèse n'a pas encore été publiée, mais les autres contributeurs de MetaCoq et moi-même y œuvrons actuellement.

27 : Une définition – dûe à Simon Boulier – d'un algorithme de typage dont la correction était établie mais pas la complétude y était déjà présent, même si j'ai eu à la modifier pour ma preuve de complétude.

Enfin le Chapitre 6 correspond à un projet que j'ai entamé dans le but d'étendre MetaCoq pour intégrer des règles η d'extensionalité à la conversion, mais qui n'a pas encore atteint le stade de la publication. J'ai en revanche présenté les difficultés qui m'y ont mené dans Lennon-Bertrand [Len22].

[Len22] : Lennon-Bertrand (2022), *À bas l'η – Coq's troublesome η-conversion*

2. Introduction

"Coq is an old man now, and it has a lot of scars."

[Har20, citing Assia Mahboubi]

This thesis belongs to the domain of type theory,[1] itself at the crossroads between computer science and mathematical logic. One of the field's goals is to give theoretical and practical foundations for software tools helping humans in constructing and verifying proofs – in the mathematical sense. Such tools are called proof assistants, and Coq, the one on which my work was mainly focused, is central in this thesis.

Over their more than 50 years of existence, proof assistants have turned into an established technology. This history is both a blessing and a curse: as the field matured, the tools have become more and more complex, making them more and more powerful, but also more and more prone to critical bugs hiding in dark corners. At a time when they are gaining traction in an increasing number of communities concerned with high trust levels, this simply cannot be. The historical solution of keeping a small, trusted kernel – the so-called De Bruijn criterion – is not enough if we wish to keep moving on and integrate new, powerful features to keep up with the needs of users.

There is a straightforward solution to this: proof assistants have been used for decades to certify programs correctness. Why could they not prove *themselves* correct? After all, if this is the gold standard we demand for software, it should apply first and foremost to the ones used to justify that trust. For the proof assistant Coq, this is the ambition of the MetaCoq project, which aims at providing a drop-in replacement for Coq's kernel that has been proven correct, even though it handles all the subtleties and quirks of said kernel. No more trusting a complex and ever-evolving implementation, trust the formally validated *proofs* instead!

But before we can hope to achieve that goal, we need a deeper study of the structures at work in the kernel. In particular, its typing algorithm is *bidirectional*, meaning that it constantly alternates between the two problems of type *inference* – finding a type for a term – and type *checking* – verifying that a type is adequate for a term. While this structure is crucial in relating the specification of the type system to its implementation, it has been rather little studied in the context of the Calculus of Inductive Constructions (CIC), the theoretical foundation of Coq – but also of the closely related Lean, Agda...

This thesis aims at filling that gap, by providing a thorough study of bidirectional CIC, formalized in the framework offered by MetaCoq project. This is a key ingredient in the first formal proof of correctness and completeness of a type-checking algorithm for a realistic proof assistant kernel. It was also able to uncover bugs in Coq's kernel that had gone unnoticed until then.

But bidirectional typing is also an interesting theoretical tool in its own right, giving a valuable form of control over computation. In particular, it is

[Har20]: Hartnett (2020), *Building the Mathematical Library of the Future*

1: If you do not know what this or any other word in this introduction means, read on! They will be explained in due time.

2.1 A Very Short History of Logic 14
2.1.1 Syllogisms 14
2.1.2 Towards a formal foundation . 15
2.1.3 The foundational crisis 15
2.1.4 Incompleteness 15
2.1.5 A satisfactory situation? . . . 16
2.2 Computers Enter the Scene . 16
2.2.1 Proof assistants 17
2.2.2 Logic, Programming and Type Theory 17
2.3 Coq and Its Kernel 19
2.3.1 The kernel 19
2.3.2 MetaCoq 19
2.3.3 Bidirectional typing 20
2.3.4 Gradual types 20
2.4 And this Thesis? 21
2.4.1 Theory of bidirectional typing 21
2.4.2 Bidirectional typing in MetaCoq 21
2.4.3 Gradual dependent types . . . 22
2.4.4 Technical contributions 22

a necessary piece in the design of a gradual extension of CIC, GCIC. Gradual typing aims at bringing to programmers both the flexibility of development offered by dynamic typing, and the strong guarantees given by static typing, in one and the same system. GCIC intends to bring that flexibility to dependently-typed programming, and, by using the power of the Curry-Howard correspondence, to proof writing. But this endeavour comes with subtle difficulties, that can only be solved in a bidirectional setting.

To replace this work in its larger context, this introduction begins with a very short history of mathematical logic (Section 2.1), which exposes the main questions of that field. Follows a presentation of the links between logic and computer science, through proof assistants (Section 2.2). Next, Section 2.3 focuses more closely on presenting the research questions I worked on: bidirectional typing, MetaCoq and gradual typing. Finally, Section 2.4 summarizes my contributions to these questions.

2.1. A Very Short History of Logic

2.1.1. Syllogisms

The main question that logic seeks to answer is that of finding criteria in order to determine if a reasoning is valid. In Western tradition, this challenge can be traced back to the Antiquity, and particularly to Aristotle's *Organon*. The main contribution of this work is to introduce the notion of syllogism. These are simple fragments of reasoning, whose validity stems from the fixed structure they follow, rather than a specific content.[2] If complex reasoning is built from assembling such syllogisms, it must necessarily be valid as a whole, since every assembled fragment is. There are two important ideas at work here.

2: The most well-known is probably the *Barbara* syllogism, and example of which is: *all humans are mortals; Socrates is human; so Socrates is mortal.*

The first is that reasoning can be valid or not, depending only on its structure, independently of its content. It can be syllogisms, but many other systems. We will come across a certain number of them in this thesis!

The second idea is that of a construction from elementary components. Starting from a set of rules we have identified as valid *a priori*, we have a means to ensure the validity of potentially very complex reasoning: it suffices to check that these can be decomposed into the base components.

3: Structural rules reasoning should obey, as those of syllogisms.

For the Greek philosophers, logic was also conceived as a means towards communication. The aim was to check one's own reasoning, but also to be able to convey it, by fixing a logical formal system.[3] A person wanting their conclusion to be accepted by others would only have to express their reasoning in a perfectly precise way in the framework of such a formal system.

From that point on, the main focus of logic as a discipline concentrates on this structure which underlies reasoning. The main challenge is to construct a formal system, adapted to a specific field of reasoning. In the case we are interested in, mathematical logic, this allows us to give a precise meaning to what constitutes a valid mathematical proof.

2.1.2. The beginning of mathematical logic: towards a formal foundation

Following Aristotle, mathematicians seized logic in order to build a formal system able to serve as a rigorous foundation for mathematics. The links between logic and mathematics go back to Greek Antiquity, but mathematical logic as a standalone discipline really established itself during the 19th century, thanks to important progress on two main aspects.

The first consisted in freeing mathematical logic from natural languages[4], unsuited to a formal description of reasoning, and to instead design a new specific form of language that could serve as a basis for mathematical reasoning. An important step here was Frege's *Begriffsschift* [Fre79], which, for the first time, gave a formal language rich enough to express mathematics satisfyingly. Its major addition was the notion of quantifier, essential to the mathematical vernacular, as they give a faithful way to account for universal[5] and existential[6] properties.

The second aimed at showing that mathematics as a whole could be reconstructed from a few simple properties. An important step was the reduction of analysis to the properties of real numbers, followed by constructions of those from arithmetic given almost simultaneously by – among others – Dedekind [Ded72] and Cantor [Can72] in 1872. Meanwhile, Peano [Pea89] proposed an axiomatization of natural numbers close to the one still used today. Finally, Cantor again proposed set theory [Can83] as a formalism expressive enough to describe all mathematical object as sets of elements.

4: By opposition with the formal languages which appear in mathematics, computer science, etc.

[Fre79]: Frege (1879), *Begriffsschrift: Eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens*

5: For instance: "Every even natural number is the sum of two prime numbers".

6: For instance: "There exists a real whose square is 2".

[Ded72]: Dedekind (1872), *Stetigkeit und Undirrationale Zahlen*

[Can72]: Cantor (1872), *Ueber die Ausdehnung eines Satzes aus der Theorie der trigonometrischen Reihen*

[Pea89]: Peano (1889), *Arithmetices principia: Nova methodo exposita*

[Can83]: Cantor (1883), *Grundlagen einer allgemeinen Mannigfaltigkeitslehre. Ein mathematisch-philosophischer Versuch in der Lehre des Unendlichen*

2.1.3. The foundational crisis of mathematics

Unfortunately, the system proposed in the *Begriffsschift* is inconsistent ! That is, it is possible to use it to prove falsity, making the logical system collapse.[7] This result, due to Russell[8] marked the opening of a crisis period. Indeed, it cast doubt upon the systems that had started to establish themselves as good candidates to serve as foundations – that of Frege, but mainly those of Cantor, which were affected by the same difficulties.

A possible solution has been suggested ten years later by Whitehead and Russell in their *Principia Mathematica* [WR13]. This colossal piece of work not only proposed a formal system avoiding the inconsistency of *Begriffsschift*. It also built a significant amount of mathematics in this system, including a construction of integers, some arithmetic, and finally real numbers.

In parallel, in the continuity of Cantor's work, Zermelo [Zer08] and others worked towards giving a version of Cantor's set theory that is consistent. This lead to what is colloquially referred to as Zermelo-Fraenkel set theory – ZF, or ZFC when the axiom of choice[9] [Zer04] is added –, which also seemed able to serve as a solid foundation for mathematics.

7: In a system where falsity is provable, all propositions are, which is known as the principle of explosion. Such a system, where everything – and its negation – is provable can obviously not serve as an adequate foundation for mathematics.

8: In a letter to Frege in 1902 the latter made made public in Frege [Fre03, Nachwort p. 253].

[Fre03]: Frege (1903), *Grundgesetze der Arithmetik*

[WR13]: Whitehead et al. (1913), *Principia Mathematica*

[Zer08]: Zermelo (1908), *Untersuchungen über die Grundlagen der Mengenlehre I*

9: An axiom very useful in numerous branches of mathematics, but which is often treated separately, as it is both less crucial than the other axioms of ZF and at the root of counter-intuitive results.

[Zer04]: Zermelo (1904), *Beweis, daß jede Menge wohlgeordnet werden kann*

2.1.4. Incompleteness

The search for a formal system adequate as a foundation for mathematics however hit a second major difficulty: Gödel's incompleteness theorem

[Göd31]: Gödel (1931), *Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme. I*

[Göd31]. It asserts that a formal system in which one can construct integers such as those of Peano – and so *a fortiori* any system rich enough to serve mathematician's needs – cannot prove its own consistency.[10] Thus, no formal system can serve as a basis for mathematics with a formal certitude as to its adequacy. Indeed, as we cannot prove the consistency of the system in itself, it could very well turn out to be inconsistent, ruining all the efforts put into its use – just like what happened with Frege's *Begriffsschift*. And if we were to use a second system to prove the first consistent, we would only shift the prolem: now we rely on the consistency of the second system.

10: Unless the system is inconsistent, in which case it can prove *everything*, by virtue on the explosion principle, including its own consistency... and inconsistency!

A consequence of this theorem is that a system rich enough to found mathematics is necessarily incomplete.[11] Thus, in what follows, I will never refer to truth in an absolute sense – which could only be meaningful in a complete system where every statement is true or false –, but only about provability *relatively to a given system*.

11: This means that there exist independent statements, that is assertions which cannot be proven, and whose negation cannot be proven either. The consistency of the system under consideration is one example of such a statement.

2.1.5. A satisfactory situation?

Despite the difficulties put into light in the beginning of the 20th century, the research in mathematical logic reached a somewhat satisfactory situation a few decades later. First, ZFC is a reasonable formal system on which mathematics can be founded. Moreover, the mathematical community is overall convinced it would be *theoretically* possible to write down all mathematics using ZFC. This is enough for most of its members, even if those who attempt to actually give it a try, in the vein of the *Principia Mathematica*, are quite few.

12: That is, effectively expressed in a fixed formal system.

In *practice*, however, things are very different. The human development and verification of formalized mathematics[12] seems both impossible, and unnecessary. On the one hand, it would demand a considerable effort, because such mathematics would require an extremely high level of precision, both from the author of the formal proof and from the reader. At the same time, this would not significantly reduce the risk of errors. It would indeed be very hard for humans to check that some reasoning doubtlessly follows the rules of the system: a tiny error can easily creep inside thousands of pages of formal reasoning. Finally, describing mathematics in this way would drown the vital mathematical intuitions, making communication sterile.

If we wish to make formal mathematics practicable, and benefit from the guarantees they bring while eliminating these crippling defaults, we thus need new tools.

2.2. Computers Enter the Scene

A new element however radically modifies the previous situation: the advent of computers. Indeed, computer science provides new tools, making formalized mathematics both possible and attracting.

2.2.1. Proof assistants

Computers excel where humans are weak: their speciality is to treat large volumes of information in a very precise way, exactly the kind of needs brought up when manipulating formalized mathematics. Therefore, already at the beginning of the 70s,[13] software tools, collectively called *proof assistants*, start to appear, that are dedicated to writing and verifying formal proofs. Through the formalization of proofs and the verification by computers that they actually follow the rules of the underlying logical system, proof assistants open the door to a level of trust much higher than that allowed by "informal" proofs. Renowned mathematicians, such as Voevodsky [Voe10], Hales [Hal12, Preface, p. xi], or Scholze [Sch21] have indeed turned to proof assistants, particularly in order to lift uncertainties regarding the solidity of their own work.

Moreover, proof *assistants* are not simply proof checkers: beyond verification, they supply users with a large range of tools to ease the conception of formal proofs. These tools allow users to write proofs at a high level, and in an interactive manner,[14] leaving it to the proof assistant to construct the formal proofs. They range from simple facilities, such as the possibility to visualize the structure of proofs, or the tracking of hypotheses, to much more ambitious techniques.

Indeed, computer science lets us automatize entire parts of proof writing, for instance through the use of tactic languages [Del00], with which one can program proof generation. In addition, the automatic construction of proofs is a research field by itself, and the question of its integration intro proof assistants is an active topic [Bla+16; Eki+17]. Computer science has also proven its worth in the setting of mathematical computations (computer algebra systems, numerical analysis), and here again promising interactions with proof assistants are starting to arise [LW22; MMS19].

Finally, if the use of software eases the writing of proofs, proof assistants conversely open new possibilities for programming. They indeed offer a natural framework to describe in the same place the source code of a program, its specification, and the formal proof that the former fulfils the latter. This way, we can *prove* that the program runs correctly, without encountering any bugs. This mathematical certainty is much more reliable than any test set! In this field, numerous projects have already achieved large scale programs, entirely proven correct: compiler for the C language [Käs+17], implementation of the Https protocol [Bha+17], differential equations solving [Imm18]...

2.2.2. Logic, Programming and Type Theory

In order to work, proof assistants must be founded on a formal system, corresponding to the "rules" of the mathematical "game" they are supposed to enforce. Thus, they require a renewed study of mathematical logic, but with the practical aim of building tools that are at the same time powerful and easy to use. There are multiple families of proof assistants, based on very different formal systems. The one I am interested in in this thesis relies on the Curry-Howard correspondence and dependent type theory. The proof assistant Coq [Coq22a], which is at the heart of my work, belongs to this family.

13: With systems like Automath [dBru70], or Mizar [Rud92].

[dBru70]: de Bruijn (1970), *The mathematical language AUTOMATH, its usage, and some of its extensions*

[Rud92]: Rudnicki (1992), *An overview of the Mizar project*

[Voe10]: Voevodsky (2010), *Univalent foundations project*

[Hal12]: Hales (2012), *Dense Sphere Packings: A Blueprint for Formal Proofs*

[Sch21]: Scholze (2021), *Half a year of the Liquid Tensor Experiment: Amazing developments*

14: In most modern proof assistants, the final proof is built as the result of an exchange between the programmer and the tool, rather than written as a single block.

[Del00]: Delahaye (2000), *A Tactic Language for the System Coq*

[Bla+16]: Blanchette et al. (2016), *Hammering towards QED*

[Eki+17]: Ekici et al. (2017), *SMTCoq: A plug-in for integrating SMT solvers into Coq*

[LW22]: Lewis et al. (2022), *A Bi-Directional Extensible Interface Between Lean and Mathematica*

[MMS19]: Mahboubi et al. (2019), *Formally Verified Approximations of Definite Integrals*

[Käs+17]: Kästner et al. (2017), *Closing the Gap – The Formally Verified Optimizing Compiler CompCert*

[Bha+17]: Bhargavan et al. (2017), *Everest: Towards a Verified, Drop-in Replacement of HTTPS*

[Imm18]: Immler (2018), *A Verified ODE Solver and the Lorenz Attractor*

[Coq22a]: Coq Development Team (2022), *The Coq Proof Assistant*

If one compares a computer program with a text in a natural language, *types* are a kind of equivalent of grammatical categories. However, contrarily to natural languages, these types are conceived at the same time as the programming language, in order to mirror properties of the objects it manipulates. Their first use is to detect manifest errors. For instance, if a procedure intended for an object of type "image" is applied to an object of type "character string", an error can be reported to the programmer.[15] But types are very versatile, and their capacity to encode properties of the underlying programs can be used for compilation, documentation, and many other applications. In our framework, for instance, types correspond to the validity of a logical reasoning.

This idea is that of the *Curry-Howard correspondence*.[16] Rather than a precise theorem, it is more of a very general concept, according to which two worlds closely resemble each other: on the one hand, that of logic and proofs, on the other that of programs and their types.

A short example says more than a long abstract talk, so let's look at the correspondence at work in Figure 2.1, in the form of inference/typing rules: each bloc presents a rule, with above the bar the hypotheses, and below the conclusion. The first three rules govern the logical conjunction "and", written $\wedge$. The first means that to deduce the proposition $A \wedge B$ ("A and B"), it is enough to deduce A and B taken individually. Conversely, if we have as hypothesis $A \wedge B$, then we can deduce both A (second rule), and B (third rule). The last three rules govern typing[17] for the pair type $A \times B$. A pair (a, b) built from a first object a of type A and a second object b of type B has type $A \times B$. Conversely, if p is a pair of type $A \times B$, then we can retrieve its first component $p.1$, which is of type A, and its second $p.2$, of type B. If we erase the terms[18] of the bottom rules, we obtain *exactly* the rules above! Thus, the programming construct of pairs corresponds to the logical concept of conjunction.

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

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

$$\frac{p : A \times B}{p.1 : A} \qquad \frac{p : A \times B}{p.2 : B}$$

Figure 2.1. Inference rules for conjunction and typing rules for pairs

This extends well beyond the specific case of conjunction, in a general correspondence between, on one side, logical propositions and their proofs, and, on the other, types and programs. We can see properties as types, and a proof of a given property as a program of the corresponding type – or the other way around! Beyond a simple analogy between formalisms of different origins, this correspondence is a powerful tool to establish a dialogue between two worlds. In particular, it relates two *a priori* quite distant problems: checking that a proof is valid, and checking that a term is well-typed. In both cases, it amounts to checking that a construction – program on one side, proof on the other – respects a set of formal rules guaranteeing it is well-formed.

The Curry-Howard correspondence is therefore ideal to serve as a foundation for proof assistants, since it gives access, when studying formal logical systems, to the rich literature on programming languages, in particular on the theory and implementation of types. In this framework, the *dependent type systems* are a specific family of type systems, whose main characteristic is the ability for types to depend on terms. The archetypical example from the point of view of programming is the type $\mathbf{Ve}\,(A, n)$ of vectors of length n. These are lists that contain exactly n elements of type A – with n a natural number. This type depends on n, in the sense that the type's inhabitants differ depending on the integer's value. From the point of view of logic, this dependency corresponds to quantification: if we wish to express

15: A well-known slogan due to Milner [Mil78] claims that "Well-typed programs cannot go wrong."

[Mil78]: Milner (1978), *A theory of type polymorphism in programming*

16: Made explicit for the first time in informal notes by Howard dating back to 1969, but published only much later [How80], themselves based upon previous remarks by Curry [CFC58].

[How80]: Howard (1980), *The Formulae-as-Types Notion of Construction*

[CFC58]: Curry et al. (1958), *Combinatory Logic*

17: Written using a colon.

18: In the context of type theory, we often talk about *terms* instead of programs, but the two are synonyms.

a universal property "for all x, the property $P(x)$ holds", then we need the property P to depend on x. Thanks to this ability to express quantification, dependent types are rich enough to serve as foundations for mathematics.

2.3. Coq and Its Kernel

Let us now focus a bit more on the proof assistant which we will consider mainly in this thesis: Coq.

2.3.1. The kernel, cornerstone of the system

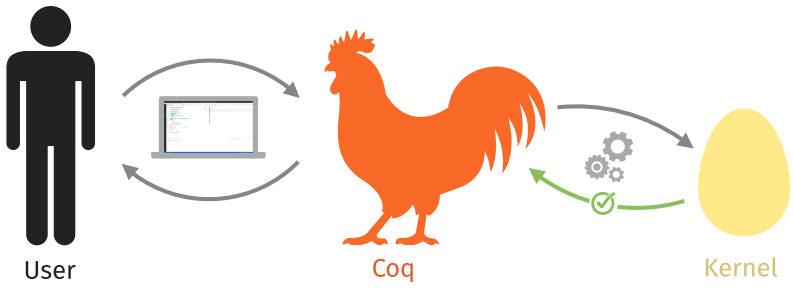

Figure 2.2. Coq's schematic architecture

Coq is based on the Curry-Howard correspondence: proofs are seen as programs, in a language called Gallina, and their verification is done using an algorithm close to those used for types in conventional languages. However, if, in the first versions from the 80s, Coq proof were mostly written directly in Gallina, it is no longer the case at all. The reason is that the major part of the tool in its current versions aims at helping the user in generating a correct proof. It is a true proof *assistant*! The way Coq works is illustrated in Figure 2.2 : the user interactively exchanges with Coq, which uses this interaction to generate a proof term. This proof term is then sent to a very specific part of the tool, called the *kernel*. This is the part implementing the type-checking algorithm, and thus responsible for ensuring that the proof terms built interactively are correct. The kernel is thus the crucial part of Coq, because it is the one – and only – ultimately responsible for proof-checking. This architecture, which clearly isolates the critical part of the system, is called *De Bruijn criterion* [BG01], in tribute to one of the pioneer of proof assistants.

[BG01]: Barendregt et al. (2001), *Proof-Assistants Using Dependent Type Systems*

If the rest of the ecosystem has grown much more than the kernel since the beginning, the latter has also evolved, becoming gradually more complex. And, as any other software development, it is not safe from bugs.[19] These are in general hard to exploit for a user, even more so without noticing. But still, they exist, and since the kernel tends to get more and more complex, they are likely to continue appearing.

19: The magnitude is that of one critical bug found every year, a list is maintained at the following address: https://github.com/coq/coq/blob/master/dev/doc/critical-bugs.

2.3.2. MetaCoq, a formalization in Coq, for Coq

If we wish to guarantee a trust level as high as possible in the kernel, we must resort to new ideas. This is what the MetaCoq project is all about. The idea is simple: use Coq itself to certify the correctness of its kernel.

More precisely, the first step is to describe formally the type system on which the kernel is based, and to show its theoretical properties. This is already a difficult endeavour: in order to ease its use, Coq's type theory incorporates a lot of complex features.

Once this meta-theory is established, the second step consists in implementing a type-checking algorithm as close as possible to the one of the kernel, directly in Gallina[20]. We show, while defining the algorithm, that it is indeed *correct*[21] and *complete*[22].

Finally, in a third step, we extract out of this certified Gallina program another more efficient program, by erasing the content related to the proof of correctness, in order to keep only the algorithmically relevant one. This extraction is a complex but cructial step if we wish to replace the current kernel while keeping a reasonable efficiency. Therefore, we also prove that said extraction is correct,[23] once again by programming it in Gallina.

20: Indeed, thanks to the Curry-Howard correspondence, Gallina is not only a proof language, but also a true programming language!

21: If the algorithm claims that a term is well-typed, then it is the case.

22: The algorithm answers positively on all well-typed programs.

23: Meaning that it preserves the semantics of programs.

2.3.3. Checking, inference and bidirectional typing

While proving the correctness of the type-checker is relatively easy once the meta-theoretical properties of the type system have been established, completeness is harder. In order to prove it, it is very useful to go through an intermediate specification, which is more structured than the theoretical one. In particular, it is important to separate two close but distinct questions: on the one side, type-checking, where we *check* that a term indeed has a given type; on the other side, inference, where we try and *find* a type for a term, if such a type exists. The typing algorithm of Coq's kernel is *bidirectional*, meaning that it alternates constantly between these two processes when it checks that a term is well-typed. Describing this bidirectional structure independently of the algorithm allows for a clear separation between, on the one side, its equivalence with the original specification, and, on the other, the part purely dedicated to implementation questions.

In the specific case of dependent types, even if present in type-checking algorithms since the origin – see *e.g.* [Hue89] –, bidirectional typing has been relatively little studied. However, beyond its strong relation to algorithms, this approach also presents theoretical advantages: its more constrained structure makes it easier to obtain properties that are difficult to obtain in the standard context.

[Hue89]: Huet (1989), *The Constructive Engine*

2.3.4. Gradual types: some flexibility in a desperately static world

There are two main approaches to program type-checking. In the static approach,[24] types are verified prior to the execution, whereas, in the dynamic approach, the well-typedness of operations is verified on the fly during that same execution. The dynamic discipline is more flexible, as it checks exactly what is necessary for the good execution of a program. The strictness of static typing, conversely, allows for error detection earlier in the development, and imposes invariants useful to optimize compilation or execution.

Instead of opting exclusively for one of the two approaches, *gradual typing* [Sie+15] aims at integrating the static and dynamic disciplines in one and

24: On which Coq is based.

[Sie+15]: Siek et al. (2015), *Refined Criteria for Gradual Typing*

the same language. The main idea is to have a first pass of verification before the execution, as in static typing, while leaving the possibility to defer parts of the verification to the execution, as in dynamic typing. This gives access to a whole spectrum of options, from a rigid completely static discipline to a flexible dynamic one. It particularly allows for a fine-grained, local choice of how each part of a program is type-checked. One can thus evolve the discipline during software development, benefiting from the flexibility of dynamic typing in early phases, and from the guarantees of static typing later on.

As the case of MetaCoq illustrates, Coq can be used as a true programming language. Even better: its type system can express very complex properties of programs, and thus verify even before their execution that the code indeed enforces them. Sadly, these reinforced constraints can turn against the user, by making the early development phase more difficult. Indeed, nobody writes correct code on the first try, and it would often be nice to temporarily lift the strong guarantees of typing to facilitate experimentation. The idea then is to take inspiration from gradual typing, in order to pave the way for a more flexible logical or software development. Once again, the Curry-Howard correspondence is at work, since we adapt concepts from the world of programming languages to the logical one.

2.4. And this Thesis?

My doctoral work itself is centred around bidirectional typing, under three main aspects, corresponding to the three parts of this thesis. They are preceded by Chapter 3, which introduces the main technical notions used in what follows.

2.4.1. Theory of bidirectional typing

The first part (Bidirectional Calculus of Inductive Constructions) proposes to – partially – fill the theoretical gap around bidirectional typing for dependent types. More precisely, it contains a proof of equivalence between the standard presentation of CIC in the literature, and a bidirectional one. Chapter 4 presents the main ideas in a relatively simple setting, in order to ease the exposition. Chapter 5 shows how to extend them to a more realistic setting, close to the type theory implemented in Coq. Finally, Chapter 6 focuses on the particular status of conversion[25], and the links between recent work on this subject and bidirectional typing.

25: This crucial notion allows the integration into dependent type theory of the notion of computation of programs.

2.4.2. Bidirectional typing in MetaCoq

The second part of the thesis (A Certified Kernel for Coq, in Coq) focuses on the MetaCoq project, and especially the formalization, in Coq, of the ideas presented in the first part. Chapter 7 gives a general overview of the project, while Chapter 8 concentrates more specifically on the proof that the kernel implemented in MetaCoq fulfils its specification.

2.4.3. Gradual dependent types

Finally, the third and last part (Bidirectional Elaboration for Gradual Typing) presents my work in the area of gradual types. Since dependent types already form complex systems, their adaptation to the gradual approach is particularly delicate. A summary of the possibilities and issues is presented in Chapter 9. An interesting point of emphasis is that the usual presentation of dependent types turns out to be unsuited, as it is too flexible. The additional structure provided by bidirectional typing is key to solve this issue. It is also relevant to present the type-directed elaboration of terms from a source language to a target one, an important characteristic shared by all gradual languages. The use of a bidirectional elaboration, and the properties it allows us to obtain, are described in Chapter 10. Finally, Chapter 11 describes follow-up work complementing that of Chapter 10, but which is not directly linked to bidirectional typing.

2.4.4. Technical contributions

My doctoral work started with the study of gradual dependent types. I contributed, together with Kenji Maillard, Nicolas Tabareau and Éric Tanter, to Lennon-Bertrand et al. [Len+22], where we study a gradual extension to the Calculus of Inductive Constructions. My main technical contribution corresponds to Chapter 10. The precise literature review and the impossibility theorem of Chapter 9 it leads to also comes from this publication. The second technical part of Lennon-Bertrand et al. [Len+22], in which I participated but whose main author is Kenji Maillard, as well as a second article,[26] together with the same authors and again Kenji Maillard as main investigator, correspond to Chapter 11.

[Len+22]: Lennon-Bertrand et al. (2022), *Gradualizing the Calculus of Inductive Constructions*

26: Maillard et al. [Mai+22], currently under review.

[Mai+22]: Maillard et al. (2022), *A Reasonably Gradual Type Theory*

This work having shown the relevance of a bidirectional dependent type system and the relative scarceness of results on the subject, I focused more closely on it, both on paper and by means of a formalization based on MetaCoq. This led to a second publication [Len21], and corresponds to Chapters 4 and 5 for the theoretical part, and Section 8.1 for the formalized proof of equivalence between bidirectional and undirected typing. The completeness bug in the kernel of Coq found during this formalisation, together with the impact of this discovery on the implementation of Coq is presented in Sozeau, Lennon-Bertrand, and Forster [SLF22].

[Len21]: Lennon-Bertrand (2021), *Complete Bidirectional Typing for the Calculus of Inductive Constructions*

[SLF22]: Sozeau et al. (2022), *The Curious Case of Case: Correct & Efficient Representation of Case Analysis in Coq and MetaCoq*

I then turned to the closer integration of this formalization into MetaCoq, and its use in order to prove completeness of the kernel it implements.[27] This is described in Section 8.3. I also contributed more generally to the project on various more minor points. This part of my thesis work has not been published yet, but the other contributors to MetaCoq and I are currently working on it.

27: A definition of a type-checking algorithm proven correct but not complete by Simon Boulier was already present, although I had to alter it during the completeness proof.

Finally, Chapter 6 corresponds to a project I initiated in order to extend MetaCoq to integrate extensionality η rules to conversion, but which did not reach the stage of publication yet. Yet, I presented the difficulties that led me to it in Lennon-Bertrand [Len22].

[Len22]: Lennon-Bertrand (2022), *À bas l'η – Coq's troublesome η-conversion*

The Calculus of Inductive Constructions

3.

Most of this thesis revolves around dependent type systems. Due to their complexity, there is a high number of points subject to slight variations when one tries to give a precise definition of a system. Some of these variations are unimportant, but some introduce subtle albeit large differences in the resulting systems. In this chapter we go in details over the definition of what I refer to as the Calculus of Inductive Constructions (CIC) in the rest of this thesis, where it serves as the base system. While doing so, I try to give an idea of the trade-offs involved, and of the reasons behind the choices. Quite a few of those vary during the thesis, and this is by design: there is no single better choice, instead one has to adapt to the setting.

For the impatient specialists, let me say now that with CIC, I mean an intensional type theory, with Church-style abstractions, a predicative hierarchy of universes[1] *à la* Russell, and any amount of inductive types presented by recursors. Conversion is the reflexive, symmetric, transitive and congruent closure of βι-reduction, and so in particular it is untyped.

For the others, the present chapter aims at introducing the basic systems and properties which we refer to in the rest of the text. Section 3.1 introduces the basic notions; Section 3.2 presents a first type system, the Calculus of Constructions (CC_ω), the purely functional core all our systems rely on; Section 3.3 defines the main notions of conversion and reduction encountered in the rest of the thesis; Section 3.4 introduces the main properties our systems should satisfy; Section 3.5 adds inductive types to CC_ω to build CIC; finally Section 3.6 discusses the extra additions to go from CIC to the Polymorphic, Cumulative Calculus of Inductive Constructions (PCUIC), a faithful model of the type theory implemented by the kernel of Coq.

3.1 Terms and Types 23
3.2 Functional Core: CC_ω 24
3.2.1 Functions and applications . . 24
3.2.2 Universes 25
3.3 50 Shades of Conversion . . . 26
3.3.1 Declarative conversion 28
3.3.2 Algorithmic conversion 28
3.4 The Good Properties 30
3.4.1 Stability under basic operations 30
3.4.2 Properties of types 31
3.4.3 Subject reduction 32
3.4.4 Progress 33
3.4.5 Normalization 34
3.5 Adding Inductive Types: CIC 35
3.5.1 Booleans 36
3.5.2 Recursion 36
3.5.3 Parameters 38
3.5.4 Indices 39
3.5.5 The Calculus of Constructions 40
3.6 Beyond CIC: PCUIC 41
3.6.1 Cumulativity 41
3.6.2 The sort of propositions 42
3.6.3 Local definitions 42
3.6.4 Global environments 43
3.6.5 Enhanced inductive types . . . 44
3.6.6 Records and co-inductive types 44

1: And only those: by default I do *not* include an impredicative sort of propositions, a feature often associated with the name CIC. I still use that name because of two characteristics that I feel sets apart the tradition around CIC in the dependent type theory literature: the definition of conversion as an *untyped* relation, and the use of Church-style abstractions. See Appendix A for a longer discussion.

3.1. Terms and Types

Throughout this chapter, type systems are defined by means of a relation $\Gamma \vdash t : T$, which reads "in the context Γ, the term t has type T". From the logical point of view, this judgement means that Γ is the list of hypothesis available to deduce the conclusion T by means of the proof t. On the programming side, it means that t is a well-formed program of type T, which uses the variables listed together with their types in Γ. Hence, Γ is a list of declarations, of the form $x : A$. We write $\cdot$ for the empty context, $\Gamma, x : A$ for the extension of context Γ with the new variable $x : A$, and $(x : A) \in \Gamma$ to denote that the declaration $x : A$ appears in the context Γ.

This typing relation itself is defined by means of inference rules, such as Rule Var opposite. The way to read this rule is that the judgement underneath the line follows from the one above, *i.e.* from $(x : A) \in \Gamma$ and $\vdash \Gamma$ – a judgement that we will soon define asserting that the context Γ is well-formed – we can deduce $\Gamma \vdash x : A$. When objects appear in the hypothesis but not the conclusion, they are implicitly universally quantified. Once a set of such inference rules is fixed, typing is defined as the least relation closed by those rules. Equivalently, a judgement such as $\Gamma \vdash t : T$ holds whenever

$$\textsc{Var}\ \frac{\vdash \Gamma \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A}$$

Figure 3.1a. Typing rule for a variable

we can build a tree whose nodes are instances of the inference rules, and whose root is the judgement in question. A general setting for this kind of definitions of type systems can be found in Bauer, Haselwarter, and Lumsdaine [BHL20], but in our case we restrict to this level of informality for the time being.[2]

[BHL20]: Bauer et al. (2020), *A general definition of dependent type theories*

2: In Part 'A Certified Kernel for Coq, in Coq', however, such judgements are formalized as inductively defined propositions.

As we have already introduced variables, a word on those as well. Variables are difficult to account for precisely, because of issues like shadowing – a conflict between two variables with the same name – or *α-equality* $=_\alpha$ – the identification between two terms only differing on variable names. There are multiple techniques to solve these issues – see the many solutions to the POPLMark Challenge [Ayd+05] –, but we again treat these in an informal way, assuming there is no shadowing whatsoever and identifying α-equal terms when needed.[3]

[Ayd+05]: Aydemir et al. (2005), *Mechanized metatheory for the masses: the POPLmark challenge*

3: A precise treatment is again given in Part 'A Certified Kernel for Coq, in Coq', where we use De Bruijn variables.

A final important building block of all our type theories is *substitution*, that we write $t[x \coloneqq u]$. This meta-operation replaces every occurrence of x in t by the term u. Once again, we treat this operation informally, assuming it never creates shadowing – what is sometimes called "capture-avoiding" substitution. It is sometimes useful to substitute multiple variable at once in parallel, which we write $t[x_1 \coloneqq u_1, \dots, x_n \coloneqq u_n]$.

3.2. Functional Core: CC_ω

Let us now turn to the core of CIC, namely the *Calculus of Constructions* (CC_ω). Through the Curry-Howard correspondence, it is both a typed form of λ-calculus – *i.e.* a kind of purely functional programming language – and a minimal form of logic – only containing universal quantification and implication. Since its introduction by Coquand and Huet [CH88], it has been the subject of intense theoretical study, modifications, and extensions, so let us fix what we exactly mean with "CC_ω".

[CH88]: Coquand et al. (1988), *The calculus of constructions*

$$\frac{\Gamma \vdash A : \square \qquad \Gamma, x : A \vdash t : T}{\Gamma \vdash \lambda\, x : A.\, t : A \to T}$$

$$\frac{\Gamma \vdash f : A \to T \qquad \Gamma \vdash u : A}{\Gamma \vdash f\ u : T}$$

Figure 3.1b. Typing for non-dependent functions

$$\textsc{Abs}\ \frac{\Gamma \vdash A : \square \qquad \Gamma, x : A \vdash t : T}{\Gamma \vdash \lambda\, x : A.\, t : \Pi\, x : A.\, T}$$

$$\textsc{App}\ \frac{\Gamma \vdash f : \Pi\, x : A.\, T \qquad \Gamma \vdash u : A}{\Gamma \vdash f\ u : T[x \coloneqq u]}$$

Figure 3.2a. Typing for dependent functions

3.2.1. Functions and applications

Let us start with the basic terms: functions and applications.

Functions, also called λ-abstractions, are written $\lambda\, x : A.\, t$. This corresponds to the mathematical notation $x \mapsto t$: the body t of the function is a term that might contain the variable x, and the constructor λ abstracts over that variable to build a function. Conversely, function application is denoted by simple juxtaposition, as in $t\ u$. The type of functions is written →, as in ordinary mathematics. You can see those at work in Figure 3.1b: an abstraction builds a term of arrow type, and application needs its function to be of an arrow type, whose domain must moreover correspond to the type of the argument. The side-condition $\Gamma \vdash A : \square$ ensures that the annotation is a valid type, we will introduce it shortly. Logically, those rules make sense if → is read as implication: if from a hypothesis A one can deduce T, then $A \to T$ holds; conversely if $A \to T$ and A both hold, then T does as well.

These arrow types, however, are not as expressive as one could hope for. Remember that we are in the realms of dependent types, so not only t might mention x, but also T. For instance, T might be something like "x is even". In such a case, we need to record that dependency, which is the point of

Π-types – or dependent function types –, shown in Figure 3.2a. Seen as function types, they record the fact that the codomain might vary depending on the argument. This is reflected in the typing rule for application: since the codomain T might depend on x, the type of the application $f\ u$ is T *specialized at the argument* u, using substitution. Seen on the logical side, Π-types correspond to universal quantification $\forall\, x\colon A.\ T(x)$. Indeed, if one can show that $T(x)$ holds for an unspecified x, then it must hold for all $x\colon A$ – this is Rule Abs. Conversely, if T holds for all $x\colon A$, then one can deduce $T(u)$ for any specific $u : A$ – this is Rule App. The rules of Figure 3.1b are just a special case of those, in the case where the codomain T does not depend on the variable x, and we use this convention throughout the thesis: $A \to T$ is shorthand for $\Pi\, x\colon A.\ T$ when T does not mention x.

One last thing to note about our functions is that they record the type of their domain – what is called *Church-style* abstraction [Bar92, Section 3]. There is an alternative – the *Curry-style* abstractions –, that does not do so, simply using $\lambda\, x.\ t$ for functions. This difference becomes important as soon as one looks at the bidirectional structure. Indeed, the annotation is required if one wants to infer types for functions, rather than barely checking them. The Curry-style option is sensible though, see for instance the implementation of the proof assistant Agda [Nor07, p. 19], Abel, Öhman, and Vezzosi [AÖV17] or McBride [McB22]. In the end, this is really a design choice between being able to infer a type for any term, or requiring annotations that in a lot of cases are useless. In this thesis we stick with the approach used in Coq, and annotate our abstractions.

[Bar92]: Barendregt (1992), *Lambda Calculi with Types*

[Nor07]: Norell (2007), *Towards a practical programming language based on dependent type theory*

[AÖV17]: Abel et al. (2017), *Decidability of Conversion for Type Theory in Type Theory*

[McB22]: McBride (2022), *Types Who Say Ni*

3.2.2. Universes

To be able to express ideas like induction principles or polymorphic functions, it is extremely useful to use functions and Π-types quantifying over types. This is what the universe $\square$ – read "Type" — is for. It is the type... of a type. This also means that the border between types and terms is not a syntactic one, because *e.g.* functions can abstract over a type. Instead, types are simply terms of type $\square$. Despite this, we still use upper case letters for terms which we want to think of as types. Such a universe is called *à la* Russell [Pal98], by contrast with universes *à la* Tarski, which regain the distinction between types and terms at the cost of a somewhat heavier treatment of types. Since we have not much use for a presentation *à la* Tarski in this thesis, we use the simpler one.

[Pal98]: Palmgren (1998), *On universes in type theory*

$$\textsc{Univ}\ \frac{\vdash \Gamma}{\Gamma \vdash \square_i : \square_{i+1}}$$

Figure 3.2b. Typing for universes

There is an important caveat regarding universes. Since the paradox exhibited by Russell in Frege's *Begriffsschift* [Fre79], logicians know that considering a set of all sets is a great source of inconsistencies. Type theory is not devoid of this issue: Girard [Gir72, Annex A] shows how having a type with itself as type is inconsistent. This inconsistency directly applies to the first dependent type system proposed by Martin-Löf [Mar72], which has a single universe $\square$ and a rule $\square : \square$. A common solution to this issue is to stratify universes into an infinite hierarchy, which gives us Rule Univ. Note how $\square$ is indexed by the *universe levels* i and $i{+}1$.

[Fre79]: Frege (1879), *Begriffsschrift: Eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens*

[Gir72]: Girard (1972), *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*

[Mar72]: Martin-Löf (1972), *An intuitionistic theory of types*

$$\Pi\textsc{Ty}\ \frac{\Gamma \vdash A : \square_i \qquad \Gamma, x\colon A \vdash B : \square_j}{\Gamma \vdash \Pi\, x\colon A.\ B : \square_{\max(i,j)}}$$

Figure 3.2c. Typing for dependent function types

Using those universes, Rule ΠTy gives the typing rule for Π-types. We can also now give a definition of the $\vdash \Gamma$ judgement, asserting that a context is well-formed, in Figure 3.2d. It simply means that all its types are indeed types. Note that in Rule Ext, we did not write down a level for the universe,

$$\textsc{Empty}\ \frac{}{\vdash \cdot}$$

$$\textsc{Ext}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A : \square}{\vdash \Gamma, x : A}$$

Figure 3.2d. Context well-formation

[HP91]: Harper et al. (1991), *Type checking with universes*

we do so to mean the existence of some unconstrained one in order to ease reading.

One last important point regarding universes is the kind of levels used. A simple solution is to rely on natural numbers (of the meta-theory), with the $+1$ and max operations interpreted by the usual ones. This is however not strictly necessary: we need levels to form a (well-founded) pre-order to avoid inconsistency, and operations such as $+1$ and max to express our typing rules, but levels could very well be something different from natural numbers. In particular, the natural number approach fixes at which exact level a particular construction is done, which is usually much more rigid than what one would wish for. A more flexible approach, introduced under the name *typical ambiguity* by Harper and Pollack [HP91], uses level expressions based on level variables, rather than numbers. This way, one can collect exactly the constraints between levels required for a term to type-check, without artificially enforcing a rigid interpretation by fixing their value to a precise number once and for all. To simplify the presentation, our default CC_ω and CIC nonetheless use natural numbers, but typical ambiguity appears at multiple points in this thesis.

3.3. 50 Shades of Conversion

$$\textsc{Conv}\ \frac{\Gamma \vdash t : T \qquad \Gamma \vdash T \cong T' : \square}{\Gamma \vdash t : T'}$$

Figure 3.2e. Conversion rule

4: This wraps up our typing rules for CC_ω, collected in Figure 3.2. The rule for non-dependent functions is not included, since the one for dependent functions subsumes it.

There is one big missing part in the picture so far. Remember we are working with dependent types, and that those can contain terms, which in turn can be seen as programs. In the case for instance of the vector type we used in the introduction – and that we are about to introduce formally –, what happens if a function expects an argument of type $\mathbf{Ve}\,(A, 3)$, but it is given as argument the output of a concatenation function, which naturally has type $\mathbf{Ve}\,(A, 2+1)$? Surely we must have a way to relate both, since after all the small program $2+1$ ought to compute 3! This is exactly what Rule Conv[4] is for: it allows to replace a type T with one that is related to it by *conversion*, written $\cong$. As usual, there are two ways to look at this relation. From the point of view of programs, it incorporates a computational aspect directly inside the type system. From the point of view of logic, it corresponds to types being the same "by definition" rather than due to some reasoning – which is why conversion is also called definitional equality or judgemental equality. In our vector example, for instance, the two types are the same by virtue of the definition of addition.

Conversion is a complex relation, arguably the most subtle part of dependent types. Consequently, there are quite different ways to present it, which in turn serve different needs. For this reason, we took care to set the typing rules of Figure 3.2 up so that nothing has to be changed in those when one definition of conversion or another is taken. The only difference is in how the relation $\Gamma \vdash T \cong T' : \square$ is defined. This way, we can treat conversion as a black box when talking about typing, making the theory modular.

A first important divide is between *typed* and *untyped* conversion. On one side, conversion is seen as an intrinsically typed relation: terms are only convertible *at a given type*. On the other, conversion is a relation between raw terms, that does not presuppose any form of typing. Figure 3.3 gives an example of the computation rule for functions in both systems. The "content" of the two rules is the same – they equate $(\lambda\, x{:}\, A.\, t)\, u$ and $t[x := u]$ – only

$\boxed{\vdash \Gamma}$

$$\textsc{Empty}\ \frac{}{\vdash \cdot} \qquad \textsc{Ext}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A : \square}{\vdash \Gamma, x : A}$$

$\boxed{\Gamma \vdash t : T}$

$$\textsc{Var}\ \frac{(x : A) \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Univ}\ \frac{\vdash \Gamma}{\Gamma \vdash \square_i : \square_{i+1}} \qquad \Pi\textsc{Ty}\ \frac{\Gamma \vdash A : \square_i \qquad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Pi\, x : A.\, B : \square_{\max(i,j)}}$$

$$\textsc{Abs}\ \frac{\Gamma \vdash A : \square \qquad \Gamma, x : A \vdash t : T}{\Gamma \vdash \lambda\, x : A.\, t : \Pi\, x : A.\, T} \qquad \textsc{App}\ \frac{\Gamma \vdash f : \Pi\, x : A.\, T \qquad \Gamma \vdash u : A}{\Gamma \vdash f\ u : T[x := u]}$$

$$\textsc{Conv}\ \frac{\Gamma \vdash t : T \qquad \Gamma \vdash T \cong T' : \square}{\Gamma \vdash t : T'}$$

Figure 3.2. Collected typing rules for CC_ω

the side-conditions differ substantially. Typed conversion goes back to the type theory of Martin-Löf [Mar72], and is a recurring feature in its many descendants. Untyped conversion relates strongly to (untyped) λ-calculus[5] via the *Pure Type Systems* (PTS) [Bar91] literature. In this thesis, we mainly consider untyped conversion, as Coq's meta-theory has been mostly studied in that tradition. But the relation between both in the context of bidirectional typing is the main subject of Chapter 6.

[Mar72]: Martin-Löf (1972), *An intuitionistic theory of types*

5: Barendregt for instance uses the name "conversion" for the equational theory of untyped λ-calculus in his reference work on the subject[Bar85].

[Bar85]: Barendregt (1985), *The Lambda Calculus: Its Syntax and Semantics. Revised Edition.*

[Bar91]: Barendregt (1991), *An Introduction to Generalized Type Systems*

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

Figure 3.3. Example: typed and untyped β rule for conversion

A second axis is about how close the conversion relation is to an implementation. For instance, conversion should be an equivalence relation, but there are two approaches to that. The first – and most standard – one is to simply *define* conversion as an equivalence relation, by adding rules for *e.g.* transitivity, as the one of Figure 3.4. This ensures that conversion has the right properties, but means it does not directly correspond to an algorithm, as this transitivity rule cannot be directly implemented, due to the need to "invent" the middle term t'. The λ-calculus theorists have known this issue for a long time, and they have a solution: characterizing conversion by means of a reduction relation $\rightarrow^\star$, which corresponds to the idea of program evaluation [Bar85]. If this reduction is well-behaved, then two terms are convertible exactly when they reduce to the same third term. This more operational characterization is closer to what can be implemented. Turning things around, one can define conversion through reduction, and only *show* in retrospect that it has the good properties that were enforced in the first approach – typically, that it is transitive. Conversion of the first kind we call *declarative conversion*, while for the second we talk about *algorithmic conversion*.

$$\frac{t \cong\ t' \qquad t' \cong\ t''}{t \cong\ t''}$$

Figure 3.4. Example: transitivity rule for conversion

In the rest of this section we give two presentations of untyped conversion.

First, a declarative one, which we use to define CC_ω, as is standard. Second, an algorithmic one, anticipating the need for it later on in Parts References-metacoq and Referencesgradual.

3.3.1. Declarative conversion

$$\textsc{UConv}\ \frac{\Gamma \vdash T' : \square \qquad T \cong T'}{\Gamma \vdash T \cong T' : \square}$$

Figure 3.5a. Typing constraint on untyped conversion

To start our presentation of untyped conversion, let us first go back to Rule Conv. Even if we wish to describe conversion as an untyped relation, we still enforce a typing constraint in Rule Conv, in order to ensure that, whenever $\Gamma \vdash t : T$ is derivable, $\Gamma \vdash T : \square$ is as well. This is exactly the content of Rule UConv, which combines conversion with a check that the target type is indeed a well-formed type.

$$\beta\textsc{Conv}\ \frac{}{(\lambda\, x{:}\, A.\, t)\, u \cong t[x := u]}$$

Figure 3.5b. Computation rule for functions

Regarding conversion itself, the first rule is Rule βConv, which corresponds to the computational behaviour of functions: the variable of an applied λ-abstraction is replaced by the argument, using substitution.

The rest of the rules ensure conversion has the properties it should. First are the ones ensuring it forms an equivalence relation: it is reflexive (ConvRefl), symmetric (ConvSym), and transitive (ConvTrans).

$$\textsc{ConvRefl}\ \frac{}{t \cong t} \qquad \textsc{ConvSym}\ \frac{t \cong t'}{t' \cong t} \qquad \textsc{ConvTrans}\ \frac{t \cong t' \qquad t' \cong t''}{t \cong t''}$$

Figure 3.5c. Equivalence rules

A second set of rules, collected in Figure 3.5d, asserts that conversion is a congruence, meaning that it is compatible with all term formers. As for the previous three, these correspond to properties we expect from the conversion relation, that we simply declare to be true. Note that we include only congruence rules for term formers with sub-terms – we *e.g.* omit $\square$. To be exhaustive, we could have included congruence rules for all term formers, but when they have no sub-term congruence is simply a special case of Rule ConvRefl. Conversely, we could omit Rule ConvRefl altogether and derive it from congruence rules, which can be seen as a generalized form of reflexivity.

$$\frac{A \cong A' \qquad B \cong B'}{\Pi\, x{:}\, A.\, B \cong \Pi\, x{:}\, A'.\, B'} \qquad \frac{A \cong A' \qquad t \cong t'}{\lambda\, x{:}\, A.\, t \cong \lambda\, x{:}\, A'.\, t'}$$

$$\frac{f \cong f' \qquad u \cong u'}{f\, u \cong f'\, u'}$$

Figure 3.5d. Congruence rules

3.3.2. Algorithmic conversion

Before we can describe algorithmic conversion, we first need to have a look at *reduction*. Reduction is in some way an operational version of conversion. The main difference is that it is oriented, in the direction corresponding to program evaluation. It itself decomposes into three components.

The first is *top-level reduction* $\rightharpoonup$, which corresponds purely to computation, without any congruence closure properties. In CC_ω there is only the single Rule βRed.

$$\beta\textsc{Red}\ \frac{}{(\lambda\, x\colon A.\, t)\ u \rightharpoonup t[x \coloneqq u]}$$

Figure 3.6a. Top-level reduction

The second component is the congruent closure of top-level reduction, *one-step reduction* $\to^1$. It allows triggering top-level reduction exactly once, but at any position in a term. Its definition is given in Figure 3.6b. Note that while we talk about congruent closure both for conversion (Figure 3.5d) and one-step reduction, we mean a different form of closure: in the case of conversion, we demand the relation to recursively hold in all sub-terms, while for one-step reduction it is allowed in exactly one sub-term.

$$\frac{t \rightharpoonup t'}{t \to^1 t'} \qquad \frac{A \to^1 A'}{\Pi\, x\colon A.\, B \to^1 \Pi\, x\colon A'.\, B} \qquad \frac{B \to^1 B'}{\Pi\, x\colon A.\, B \to^1 \Pi\, x\colon A.\, B'}$$

$$\frac{A \to^1 A'}{\lambda\, x\colon A.\, t \to^1 \lambda\, x\colon A'.\, t} \qquad \frac{t \to^1 t'}{\lambda\, x\colon A.\, t \to^1 \lambda\, x\colon A.\, t'} \qquad \frac{f \to^1 f'}{f\ u \to^1 f'\ u}$$

$$\frac{u \to^1 u'}{f\ u \to^1 f\ u'}$$

Figure 3.6b. One-step reduction

Finally, we obtain reduction $\to^\star$ as the reflexive transitive closure of one-step reduction, see Figure 3.6c.

$$\frac{}{t \to^\star t} \qquad \frac{t \to^1 t' \qquad t' \to^\star t''}{t \to^\star t''}$$

Figure 3.6c. Reduction

We can now get to algorithmic conversion: two terms are convertible whenever they reduce to terms that are α-equal. As for declarative conversion, we impose a typing condition on the target type. Altogether, this leads to Rule AlgConv. For once, we make α-equality explicit to anticipate its replacement by more complex relations later on.

$$\textsc{AlgConv}\ \frac{\Gamma \vdash T' : \square \qquad T \to^\star U \qquad T' \to^\star U' \qquad U =_\alpha U'}{\Gamma \vdash T \cong T' : \square}$$

Figure 3.6d. Algorithmic conversion

To wrap up this section, let us backtrack for a moment on the reason why we separated the definition of reduction in three layers. This is because reduction as we defined it is somewhat too unconstrained.[6] In what follows, a recurring need is that of a deterministic notion of reduction which is able to expose a canonical term former,[7] if it exists. There is a way to do so, what is called *weak-head reduction* $\to^\star_{\mathrm{h}}$. It amounts to restricting the place in a term where top-level reduction can be used, by removing some congruence rules compared to reduction. More precisely, λ-abstractions, Π-types and universes are not reduced further, as they already are canonical forms of their types. Variables are not reduced either, since they simply cannot be. Thus, the only reduction that is allowed is in the function position of

6: In particular, it is non-deterministic.

7: This notion is formally introduced in Section 3.4.

an application, with the hope to get a λ-abstraction there that can be further reduced using top-level reduction. Following these considerations, we arrive at Figure 3.7. When we want to contrast this weak-head reduction with the previously defined one $\to^\star$, we call the latter *full reduction*.

$$\frac{t \rightharpoonup t'}{t \to_{\mathrm{h}}^1 t'} \qquad \frac{f \to_{\mathrm{h}}^1 f'}{f\ u \to_{\mathrm{h}}^1 f'\ u} \qquad \frac{}{t \to_{\mathrm{h}}^\star t} \qquad \frac{t \to_{\mathrm{h}}^1 t' \qquad t' \to_{\mathrm{h}}^\star t''}{t \to_{\mathrm{h}}^\star t''}$$

Figure 3.7. Weak-head reduction

3.4. The Good Properties

Before going further into more definitions of type systems, we should stop and consider what makes these "good". Designing type systems is a complex endeavour, and many things can go wrong. What are the properties we expect from a type system for it to give a valid notion of programming language or logic? How do we know that a type system is well-behaved? Let us go over some of these properties, and some proof techniques that can be employed to establish them.

3.4.1. Stability under basic operations

The most essential properties of a type system are its stability by basic type theoretic operations. The first is stability under renaming, which states that a context can be replaced by another one which contains at least the same variables:

Property 3.1. *Stability under renaming*

> Whenever the following conditions are met
>
> - $x_1 : A_1 \dots x_n : A_n \vdash t : T$
> - $\vdash \Delta$
> - for all i, there is a variable y_i such that $(y_i : A_i[x_1 := y_1 \dots x_n := y_n]) \in \Delta$
>
> we have that $\Delta \vdash t[x_1 := y_1 \dots x_n := y_n] : T[x_1 := y_1 \dots x_n := y_n]$.

Given the first premise, the context $x_1 : A_1 \dots x_n : A_n$ must be well-formed,[8] A_i can only depend on variables $x_1 \dots x_{i-1}$, thus we do not actually need to substitute the variables $x_{i+1} \dots x_n$ in it. However, this presentation, where the same substitution is applied to all types even if applies to variables which we know are not present in them, is easier to work with in practice.

8: This is a consequence of validity, another property we are about to see.

A direct consequence is the weakening property:

Property 3.2. *Weakening*

> Whenever $\Gamma \vdash t : T$ and $\Gamma \vdash A : \square$, it holds that $\Gamma, x : A \vdash t : T$.

A stronger notion is that of stability under substitution, which allows replacing variables by arbitrary terms.

Property 3.3. *Stability under substitution*

> For any substitution σ (function from variables to terms) such that the following hold
>
> - $x_1 : A_1 \dots x_n : A_n \vdash t : T$
> - for all x_i, we have $\Delta \vdash \sigma(x_i) : A_i[\sigma]$
>
> it is also the case that $\Delta \vdash t[\sigma] : T[\sigma]$.

These two stability properties can be proven by direct induction on the typing derivations, replacing hypotheses on the first context by hypothesis on the second. Of course, we need to state and prove similar stability properties for conversion, again by induction.

There is, however, a stronger form of stability under renaming. While not as crucial as the one above, it is still quite useful, especially to prove correctness of term manipulations, such as those operated by tactics.

Property 3.4. *Conditional stability under renaming*

> Whenever the following conditions are met
>
> - $x_1 : A_1 \dots x_n : A_n \vdash t : T$
> - $\vdash \Delta$
> - for all *i such that x_i appears in t*, there is a variable y_i such that $(y_i : A_i[x_1 \coloneqq y_1 \dots x_n \coloneqq y_n]) \in \Delta$
>
> there exists a type T' such that $\Delta \vdash t[x_1 \coloneqq y_1 \dots x_n \coloneqq y_n] : T'$.

The difference between the two is that we do not ask for all variables appearing in Γ to be present in Δ, only those that are "relevant" for t. Thus, the important consequence is the following, which allows removing unused variables from a context.

Property 3.5. *Strengthening*

> If $\Gamma, x : A \vdash t : T$ holds and x does not appear in t, there exists T' such that $\Gamma \vdash t : T'$.

Strengthening is not as easy to obtain as weakening, and there are some type theories where it fails [HB21]. In general, even if it holds – this is the case in all type theories presented in this thesis – it cannot be proven by a direct induction on the typing derivation. This is because of Rule Conv. Indeed, in that rule the target type T' might very well use the variable x, so that we do not have in general $\Gamma \vdash T' : \square$. Thus, there is a need for further reasoning to prove that such a type is never actually needed. We show in Theorem 4.8 how the bidirectional structure makes proving strengthening straightforward.

[HB21]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

3.4.2. Properties of types

A second set of properties pertain to types themselves. They are less crucial than the previous ones, but assess that the types that can be obtained for a term are well-behaved, which is often useful to have in proofs of other properties of the system – such as those in the rest of this section.

The first is validity, which asserts that both types and contexts are well-formed whenever they appear in a typing derivation.

Property 3.6. *Validity*

> Whenever $\Gamma \vdash t : T$, we have $\vdash \Gamma$ and $\Gamma \vdash T : \square$.

We set up CC_ω so that it satisfies this property, but another approach – which we use in the bidirectional setting – is to remove pre-conditions such as $\vdash \Gamma$ in Rule Var or $\Gamma \vdash T' : \square$ in Rule UConv. This is possible, but in that case a lot of properties have to be prefixed with extra hypothesis of context/type well-formation.

The second property is uniqueness of types, which relates the different types of a same term.

Property 3.7. *Uniqueness of types*

> A type theory satisfies *uniqueness of types up to* a relation $\preceq$ if whenever t is well-typed in Γ,[9] there exists a type T such that $\Gamma \vdash t : T$ and for any T' such that $\Gamma \vdash t : T'$, we have $T' \preceq T$.
>
> We simply say uniqueness of types for uniqueness up to conversion.

9: *I.e.* whenever there exists S such that $\Gamma \vdash t : S$.

Note that in the case where the relation $\preceq$ is symmetric and transitive, – in particular, conversion –, uniqueness of types up to $\preceq$ simplifies to the fact that whenever $\Gamma \vdash t : T$ and $\Gamma \vdash t : T'$, we have $T \preceq T'$. However, in PCUIC we wish to replace conversion with cumulativity, which is not symmetric – it is only a pre-order –, so the more involved definition is needed.

This property is not so easy to establish, but as for strengthening the bidirectional setting gives a straightforward proof approach, see Theorem 4.5.

3.4.3. Subject reduction

We already mentioned Milner's slogan that "*Well-typed programs cannot go wrong.*" In our context, this means that if a term is well-typed, its reduction – which corresponds to program evaluation –, should be well-behaved. This well-behaviour is separated into multiple properties, the first of which is subject reduction, which asserts that typing is preserved by reduction.

Property 3.8. *Subject reduction*

> If $\Gamma \vdash t : T$ and $t \to^\star t'$, then also $\Gamma \vdash t' : T$. This property is also called *preservation*.

To show that reduction preserves typing, it suffices to show that one-step reduction does, by a simple induction. Moreover, using stability under substitution, this further reduces to top-level reduction preserving typing. But how do we show this?

Suppose we have a β-redex such that $\Gamma \vdash (\lambda\, x{:}\, A.\ t)\ u : T$. Analysing the typing derivation, we can conclude there exists A', B and B' such that

- $\Gamma, x{:}\, A \vdash t : B$

- $\Pi\, x{:}\, A.\, B \cong \Pi\, x{:}\, A'.\, B'$
- $\Gamma \vdash u : A'$
- $B'[x \coloneqq u] \cong T$

If we were able to conclude that $A \cong A'$ and $B \cong B'$, we could deduce $\Gamma \vdash u : A$, then using stability under substitution we would get $\Gamma \vdash t[x \coloneqq u] : B[x \coloneqq u]$, which would finally lead to $\Gamma \vdash t[x \coloneqq u] : T$ using stability of conversion under substitution and transitivity of conversion. Thus, the key property is the following:

Property 3.9. *Injectivity of function types*

> Whenever $\Pi\, x{:}\, A.\, B \cong \Pi\, x{:}\, A'.\, B'$, we have $A \cong A'$ and $B \cong B'$.

In the more general setting of CIC or PCUIC, we do not have only Π-types. Thus, we more generally talk about *injectivity of type constructors*.

For declarative conversion, transitivity is trivial, but injectivity of function types is not so easy. Indeed, due to transitivity we could have

$$\Pi\, x{:}\, A.\, B \cong T_1 \cong \ldots T_n \cong \Pi\, x{:}\, A'.\, B'$$

where the T_i have no reason to be Π-types, and so it is not so easy to relate A and A'. Conversely, for algorithmic conversion, injectivity of function types is rather straightforward by induction on reduction and α-equality, but transitivity is hard to show. Thus, in both cases subject reduction is not direct. The main missing property, which allows proving equivalence of both notions of conversion, and consequently subject reduction for either one of the corresponding notions of typing, is confluence of reduction.

Property 3.10. *Confluence*

> If $t \to^\star t_1$ and $t \to^\star t_2$ hold, then there exists some t'' such that $t_1 \to^\star t''$ and $t_2 \to^\star t''$.

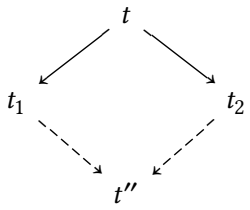

Figure 3.8. Confluence, as a diagram

This is a very widely studied property in the context of rewriting systems. A nice proof technique relies on the definition of a notion of parallel reduction [Tak95].

[Tak95]: Takahashi (1995), *Parallel Reductions in λ-Calculus*

3.4.4. Progress

Subject reduction ensures that when a term reduces, this reduction is type-preserving. The second important property linked to reduction characterizes which terms reduce. To state it, we first need to define the nm and ne predicates, characterizing respectively *normal forms* and *neutral forms*. The inductive rules for those are given in Figure 3.9. The idea is that neutral forms are those terms which are stuck on a variable, which blocks further computation because it is not a λ-abstraction. Normal forms are either neutrals or *canonical forms*,[10] which have finished computing. For instance, a λ-abstraction is the canonical form for a function. What progress says is that these forms accurately characterize well-typed terms which do not reduce.

$$\frac{}{\mathrm{nm}\ \square} \qquad \frac{\mathrm{nm}\ A \qquad \mathrm{nm}\ B}{\mathrm{nm}\ \Pi\, x{:}\, A.\, B}$$

$$\frac{\mathrm{nm}\ A \qquad \mathrm{nm}\ t}{\mathrm{nm}\ \lambda\, x{:}\, A.\, t} \qquad \frac{\mathrm{ne}\ t}{\mathrm{nm}\ t}$$

$$\frac{}{\mathrm{ne}\ x} \qquad \frac{\mathrm{ne}\ f \qquad \mathrm{nm}\ u}{\mathrm{ne}\ f\ u}$$

Figure 3.9. Normal and neutral forms

10: Alternatively called values.

Property 3.11. *Progress*

> For every well-typed term t, either $\mathrm{nm}\, t$ holds, or there is some t' such that $t \to^1 t'$.

To prove progress, one can again resort to induction on the typing derivation. The key point is to characterize the normal forms at a given type, by proving that they are either neutral forms, or canonical form *of the right kind*. For instance, if f is a normal form and has a function type, then it must be either a neutral, or a λ-abstraction. Then, if f is applied to u, then in the first case $f\ u$ is a neutral – and thus a normal form –, or it reduces further, by a β step.

One way to understand progress – and, indeed, the origin of the name – is that well-typed terms do not get stuck: either they have finished computing, and thus satisfy nm, or they should be able to make progress by reducing further. Put together with preservation, progress can be iterated. Indeed, if a term is well-typed, it is either a normal form, or reduces to a term, which is itself well-typed by preservation, so is either a normal form or reduces, and so on. This decomposition of program safety into progress and preservation has been standard since Wright and Felleisen [WF94].

[WF94]: Wright et al. (1994), *A Syntactic Approach to Type Soundness*

Property 3.12. *Safety*

> Safety is the combination of progress and preservation. It implies that if $\vdash t : T$ and $t \to^\star v \not\to^1$, then v must be a canonical form.

3.4.5. Normalization

The last important property, and one which is rather specific to type systems in the context of proof languages, is normalization. It ensures that progress cannot be applied forever, but that evaluation always ends up reaching a normal form. The most standard way to phrase this is to say that there is no infinite reduction sequence starting from a well-typed term. This formulation, however, is constructively too weak, so we instead use a more adequate – but classically equivalent – definition, using the following accessibility predicate.

Definition 3.13. *Accessibility*

> Let R be a relation on A. An inhabitant a of A is accessible if all a' such that $a\ R\ a'$ are.

In the intuitionistic setting, this way to phrase well-foundedness is much better behaved because it does not appeal to negation. In particular, we can do constructions on all accessible terms of a given relation by means of well-founded induction, something we exploit in MetaCoq, where this is the formulation we use for normalization.

Property 3.14. *Normalization*

> Every well-typed term is accessible for one-step co-reduction ${}^1\!\leftarrow$, the inverse relation of one-step reduction $\to^1$.

Normalization, combined with progress and preservation, entails that any well-typed term eventually reduces to a normal form – which is moreover unique, by confluence. This gives a naive way to decide conversion. Even if one uses a more complex strategy, normalization is a crucial building block towards decidability of typing. Thus, it is a property of prime importance if we wish to implement a type-checker for dependent types.

Another key consequence, of normalization is that, there are some uninhabited types in the empty context, for instance $\Pi\, A{:}\,\square.\ A$. This is one way to phrase logical consistency,[11] which has the advantage that it does not put forward one particular "false" type.

11: Thanks to the principle of explosion.

Property 3.15. *Logical consistency*

> There is a type which is not inhabited in the empty context.

Indeed, there are no normal forms in the empty context at that type, and since any term of that type must reduce to such a normal form, there are none. Thus, normalization ensures our type systems are meaningful as logics, which we of course care about!

More generally, normalization entails the canonicity property for *closed terms* [12] – *e.g.* those that have no free variables, or, equivalently, that are well-typed in the empty context.

12: Terms which are not closed are called *open*.

Property 3.16. *Canonicity*

> Every term t that is well-typed in the empty context reduces to a canonical form.

There is however an issue here: since normalization entails logical consistency, it is a hard property to prove. In particular, due to Gödel's incompleteness theorem, we cannot hope to prove normalization of a type system in the logic given by that system itself... Still, there are multiple approaches to proving normalization, from the venerable reducibility method [Tai67] to the recent normalization by evaluation techniques [Abe13]. However, due to their complex character, we do not tackle such proofs of normalization directly in this thesis. Instead, we either suppose normalization when it is unavoidable, or prove it relatively to that of another, simpler theory.

[Tai67]: Tait (1967), *Intensional interpretations of functionals of finite type I*

[Abe13]: Abel (2013), *Normalization by Evaluation: Dependent Types and Impredicativity*

3.5. Adding Inductive Types: CIC

Of course, not everything in mathematics or programming is a function. Although CC_ω is powerful enough to encode many constructions,[13] such encodings are not fully satisfactory: Geuvers [Geu01] shows that it is impossible to construct[14] an encoding of natural numbers satisfying an induction principle, which is their defining characteristic! Because of such limitations of encodings, and in order to faithfully represent the use of induction in mathematics and pattern-matching in programming languages, the general class of *inductive types* has been introduced by Paulin-Mohring [Pau93].[15] Adding these to CC_ω results in *CIC*, the *Calculus of Inductive Constructions*.

13: At least if one extends its universe hierarchy with an impredicative universe.

[Geu01]: Geuvers (2001), *Induction Is Not Derivable in Second Order Dependent Type Theory*

14: In a system close to our CC_ω, but again with an impredicative universe.

[Pau93]: Paulin-Mohring (1993), *Inductive Definitions in the System Coq - Rules and Properties*

15: Earlier type theories, such as [Mar72; MS84], presented specific instances of that class, but not a general scheme.

[Mar72]: Martin-Löf (1972), *An intuitionistic theory of types*

[MS84]: Martin-Löf et al. (1984), *Intuitionistic Type Theory*

3.5.1. Booleans

Let us start with a very simple example: *booleans*. To add those to CC_ω, we need to specify three new kinds of term formers. The first is the type, that we write $\mathbf{B}$ – see Rule BoolTy. Next we need *constructors*, giving the canonical inhabitants of the type. In the case of booleans, there are two of them: the false boolean ff and the true one tt – this is Rules False and True.

$$\textsc{BoolTy}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{B} : \square_0}$$

Figure 3.10a. The type of booleans

$$\textsc{False}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathrm{ff} : \mathbf{B}} \qquad \textsc{True}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathrm{tt} : \mathbf{B}}$$

Figure 3.10b. The boolean constructors

The last one is a way to use those canonical inhabitants. For booleans, this corresponds to a conditional, taking one branch or another depending on the value of the term being used, whose typing rule is given in Rule BoolInd. We call s the *scrutinee*, P the *predicate* and $b_{\mathrm{ff}}, b_{\mathrm{tt}}$ the *branches*. As was the case for dependent functions, here also there is a generalization with respect to usual programming languages: the predicate type itself can depend on the scrutinee. The usual if-then-else conditional thus corresponds to the special case when P does not depend on the variable z. We call this ind term former *induction principle*, as one can read $\mathrm{ind}_{\mathbf{B}}(s; z.P; b_{\mathrm{ff}}, b_{\mathrm{tt}})$ as case distinction on the scrutinee: to prove that P holds for an arbitrary boolean s, it suffices to show that both $P[z \coloneqq \mathrm{ff}]$ and $P[z \coloneqq \mathrm{tt}]$ do – these are respectively proven by b_{ff} and b_{tt}. The name induction is not really suitable here because we only have base cases and no induction step, but we get those as soon as the inductive type itself is recursive. We also use the name *recursor* interchangeably with induction principle, but especially when we want to emphasize the programming point of view.

$$\textsc{BoolInd}\ \frac{\Gamma \vdash s : \mathbf{B} \qquad \Gamma, z : \mathbf{B} \vdash P : \square \qquad \Gamma \vdash b_{\mathrm{ff}} : P[z \coloneqq \mathrm{ff}] \qquad \Gamma \vdash b_{\mathrm{tt}} : P[z \coloneqq \mathrm{tt}]}{\Gamma \vdash \mathrm{ind}_{\mathbf{B}}(s; z.P; b_{\mathrm{ff}}, b_{\mathrm{tt}}) : P[z \coloneqq s]}$$

Figure 3.10c. Induction principle for booleans

One thing is still missing in this picture: computation. The extension of top-level reduction is given in Figure 3.10d – our first example of ι-reduction, the reduction of recursors on constructors. These rules pick the branch corresponding to the scrutinee, which is sensible if $\mathrm{ind}_{\mathbf{B}}$ is understood as a conditional. Declarative conversion can be extended in exactly the same way. Finally, to account for the arguments of the newly introduced term former $\mathrm{ind}_{\mathbf{B}}$, we need to add new congruence rules, see Figure 3.10e. For one-step reduction and declarative conversion, there is no subtlety, all positions behave the same. The interesting rule is the one for weak-head reduction: there is only one congruence rule, which allows for reduction of the scrutinee. This is similar to functions, where we allow reduction only in the position in the term that triggers a computation if it is a canonical form – in the case of ind, the scrutinee.

$$\iota\textsc{False}\ \frac{}{\mathrm{ind}_{\mathbf{B}}(\mathrm{ff}; z.P; b_{\mathrm{ff}}, b_{\mathrm{tt}}) \rightharpoonup b_{\mathrm{ff}}} \qquad \iota\textsc{True}\ \frac{}{\mathrm{ind}_{\mathbf{B}}(\mathrm{tt}; z.P; b_{\mathrm{ff}}, b_{\mathrm{tt}}) \rightharpoonup b_{\mathrm{tt}}}$$

Figure 3.10d. Top-level reduction for booleans (ι-reduction)

3.5.2. Recursion

Booleans are very simple, but we of course want more. The first thing to add is recursion. The simplest example is that of natural numbers, given in Figure 3.11. The rules are more verbose than those for booleans, but the general idea is very similar: Rule Nat introduces a new type, Rule Zero and Rule Succ its constructors, and Rule NatInd its induction principle.

$$\frac{s \cong s' \qquad P \cong P' \qquad b_{\mathrm{ff}} \cong b'_{\mathrm{ff}} \qquad b_{\mathrm{tt}} \cong b'_{\mathrm{tt}}}{\mathrm{ind}_{\mathbf{B}}(s; z.P; b_{\mathrm{ff}}, b_{\mathrm{tt}}) \cong \mathrm{ind}_{\mathbf{B}}(s'; z.P'; b'_{\mathrm{ff}}, b'_{\mathrm{tt}})} \qquad \frac{s \to^1 s'}{\mathrm{ind}_{\mathbf{B}}(s; z.P; b_{\mathrm{ff}}, b_{\mathrm{tt}}) \to^1 \mathrm{ind}_{\mathbf{B}}(s'; z.P; b_{\mathrm{ff}}, b_{\mathrm{tt}})}$$

$$\frac{P \to^1 P'}{\mathrm{ind}_{\mathbf{B}}(s; z.P; b_{\mathrm{ff}}, b_{\mathrm{tt}}) \to^1 \mathrm{ind}_{\mathbf{B}}(s; z.P'; b_{\mathrm{ff}}, b_{\mathrm{tt}})} \qquad \frac{b_{\mathrm{ff}} \to^1 b'_{\mathrm{ff}}}{\mathrm{ind}_{\mathbf{B}}(s; z.P; b_{\mathrm{ff}}, b_{\mathrm{tt}}) \to^1 \mathrm{ind}_{\mathbf{B}}(s; z.P; b'_{\mathrm{ff}}, b_{\mathrm{tt}})}$$

$$\frac{b_{\mathrm{tt}} \to^1 b'_{\mathrm{tt}}}{\mathrm{ind}_{\mathbf{B}}(s; z.P; b_{\mathrm{ff}}, b_{\mathrm{tt}}) \to^1 \mathrm{ind}_{\mathbf{B}}(s; z.P; b_{\mathrm{ff}}, b'_{\mathrm{tt}})} \qquad \frac{s \to^1_{\mathrm{h}} s'}{\mathrm{ind}_{\mathbf{B}}(s; z.P; b_{\mathrm{ff}}, b_{\mathrm{tt}}) \to^1_{\mathrm{h}} \mathrm{ind}_{\mathbf{B}}(s'; z.P; b_{\mathrm{ff}}, b_{\mathrm{tt}})}$$

Figure 3.10e. Congruence rules for booleans

$$\textsc{Nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{N} : \square_0} \qquad \textsc{Zero}\ \frac{\vdash \Gamma}{\Gamma \vdash 0 : \mathbf{N}} \qquad \textsc{Succ}\ \frac{\Gamma \vdash n : \mathbf{N}}{\Gamma \vdash \mathrm{S}(n) : \mathbf{N}}$$

$$\textsc{NatInd}\ \frac{\Gamma \vdash s : \mathbf{N} \qquad \Gamma, z : \mathbf{N} \vdash P : \square \qquad \Gamma \vdash b_0 : P[z := 0] \qquad \Gamma, y : \mathbf{N}, p_y : P[z := y] \vdash b_{\mathrm{S}} : P[z := \mathrm{S}(y)]}{\Gamma \vdash \mathrm{ind}_{\mathbf{N}}(s; z.P; b_0, y.p_y.b_{\mathrm{S}}) : P[z := s]}$$

$$\iota\textsc{Zero}\ \frac{}{\mathrm{ind}_{\mathbf{N}}(0; z.P; b_0, y.p_y.b_{\mathrm{S}}) \rightharpoonup b_0}$$

$$\iota\textsc{Succ}\ \frac{}{\mathrm{ind}_{\mathbf{N}}(\mathrm{S}(n); z.P; b_0, y.p_y.b_{\mathrm{S}}) \rightharpoonup b_{\mathrm{S}}[y := n, p_y := \mathrm{ind}_{\mathbf{N}}(n; z.P; b_0, y.p_y.b_{\mathrm{S}})]}$$

Figure 3.11. Natural numbers

This time said induction principle is a real one, as we can see in the second branch, where an induction hypothesis p_y on the predecessor y is available. Similarly to booleans, the induction principle reduces when its scrutinee is a constructor. But, again, since we have real recursion, a recursive call appears in the reduct of Rule ιSucc. We do not repeat the congruence rules, as they are similar to those for booleans (Figure 3.10e). The only difference is that now there is also a need for congruence rules for the term former S, since it has a sub-term.

At this point, it might be good to add a note on the way we represent constructors: we enforce them to be *fully applied*, meaning S does not make sense on its own as a term. Coq is slightly more permissive, and allows $\mathrm{S} : \mathbf{N} \to \mathbf{N}$. We forbid this, but one can always consider $\lambda\, x : \mathbf{N}.\, \mathrm{S}(x)$ instead if needed. Likewise, inductive types are also enforced to be fully applied. We also avoid using the Π and λ term formers to represent binding in the predicate and branches of constructors, rather using contexts directly. This allows for a clear separation of concerns, by reducing interactions between the functional fragment and inductive types. Coq's kernel used to rely on Π and λ abstractions to represent predicates and branches, but a version close to our presentation has recently replaced it,[16] in part due to concerns raised while working on this thesis, that are detailed in Section 5.2.

16: The exact change is documented by pull-request #13563.

$$\textsc{PairTy}\ \frac{\vdash A : \square_i \qquad \Gamma, x\colon A \vdash B : \square_j}{\Gamma \vdash \Sigma\, x\colon A.\ B : \square_{\max(i,j)}}$$

$$\textsc{Pair}\ \frac{\Gamma \vdash A : \square \qquad \Gamma, x\colon A \vdash B : \square \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : B[x \coloneqq t]}{\Gamma \vdash (t,u)_{(A,x.B)} : \Sigma\, x\colon A.\ B}$$

$$\textsc{PairInd}\ \frac{\begin{array}{c}\Gamma \vdash s : \Sigma\, x\colon A.\ B \qquad \Gamma, z\colon \Sigma\, x\colon A.\ B \vdash P : \square \\ \Gamma, y_1\colon A, y_2\colon B[x \coloneqq y_1] \vdash b : P[z \coloneqq (y_1,y_2)_{(A,x.B)}]\end{array}}{\Gamma \vdash \mathrm{ind}_\Sigma\,(s; z.P; y_1.y_2.b) : P[z \coloneqq s]}$$

$$\iota\textsc{Pair}\ \frac{}{\mathrm{ind}_\Sigma\,((t,u)_{(A,x.B)}; z.P; y_1.y_2.b) \rightharpoonup b[y_1 \coloneqq t, y_2 \coloneqq u]}$$

Figure 3.12. Inductive dependent pair type

$$\textsc{ListTy}\ \frac{\vdash A : \square_i}{\Gamma \vdash \mathbf{Li}\,(A) : \square_i} \qquad \textsc{Nil}\ \frac{\Gamma \vdash A : \square}{\Gamma \vdash \varepsilon_A : \mathbf{Li}\,(A)}$$

$$\textsc{Cons}\ \frac{\Gamma \vdash A : \square \qquad \Gamma \vdash a : A \qquad \Gamma \vdash l : \mathbf{Li}\,(A)}{\Gamma \vdash a ;;_A l : \mathbf{Li}\,(A)}$$

$$\textsc{ListInd}\ \frac{\begin{array}{c}\Gamma \vdash s : \mathbf{Li}\,(A) \qquad \Gamma, z\colon \mathbf{Li}\,(A) \vdash P : \square \qquad \Gamma \vdash b_\varepsilon : P[z \coloneqq \varepsilon] \\ \Gamma, y_1\colon A, y_2\colon \mathbf{Li}\,(A), p_{y_2} : P[z \coloneqq y_2] \vdash b_{;;} : P[z \coloneqq y_1 ;;_A y_2]\end{array}}{\Gamma \vdash \mathrm{ind}_{\mathbf{Li}}\,(s; z.P; b_\varepsilon, y_1.y_2.p_{y_2}.b_{;;}) : P[z \coloneqq s]}$$

$$\iota\textsc{Nil}\ \frac{}{\mathrm{ind}_{\mathbf{Li}}\,(\varepsilon_A; z.P; b_\varepsilon, y_1.y_2.p_{y_2}.b_{;;}) \rightharpoonup b_\varepsilon}$$

$$\iota\textsc{Cons}\ \frac{}{\begin{array}{l}\mathrm{ind}_{\mathbf{Li}}\,(a ;;_A l; z.P; b_\varepsilon, y_1.y_2.p_{y_2}.b_{;;}) \rightharpoonup \\ b_{;;}[y_1 \coloneqq a, y_2 \coloneqq l, p_{y_2} \coloneqq \mathrm{ind}_{\mathbf{Li}}\,(l; z.P; b_\varepsilon, y_1.y_2.p_{y_2}.b_{;;})]\end{array}}$$

Figure 3.13. List type

3.5.3. Parameters

A second direction for enhancement is the ability to have inductive types with parameters. The main use of this is for type operators, that is types that take other types as arguments, for instance the pair type $A \times B$ of Chapter 2. As is probably not very surprising by now, this type is a restricted instance of a more general type, the dependent pair type $\Sigma\, x\colon A.\ B$. Logically, its dependency on A means that if we see B as a property, the whole pair type describes a subset of A – those elements which validate B. The rules are given in Figure 3.12. Similarly to functions, we need an annotation on the pair constructor, for the exact same reason: we want to ensure that any term can infer a type. We also omit congruence rules, as they are again similar to those of Figure 3.10e, although now not only the pair constructor but also the type constructor Σ get their congruence rules, since both have sub-terms.

As an example which combines both recursion and parameters, we have the polymorphic list type $\mathbf{Li}$, which mainly combines what we already covered for natural numbers and pairs. The typing and reduction rules are given in Figure 3.13.

$$\textsc{EqType}\ \frac{\Gamma \vdash A : \square_i \qquad \Gamma \vdash a : A \qquad \Gamma \vdash a' : A}{\Gamma \vdash a =_A a' : \square_i}$$

$$\textsc{EqRefl}\ \frac{\Gamma \vdash A : \square_i \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathrm{refl}_{A,a} : a =_A a}$$

$$\textsc{EqInd}\ \frac{\Gamma \vdash s : a =_A a' \qquad \Gamma, y\colon A, z\colon a =_A y \vdash P : \square \qquad \Gamma \vdash b : P[y \coloneqq a, z \coloneqq \mathrm{refl}_{A,a}]}{\Gamma \vdash \mathrm{ind}_= (s; y.z.P; b) : P[y \coloneqq a', z \coloneqq s]}$$

$$\iota\textsc{Eq}\ \frac{}{\mathrm{ind}_= (\mathrm{refl}_{A,a}; y.z.P; b) \rightharpoonup b}$$

Figure 3.14. Equality type

3.5.4. Indices

There is one feature missing in the previous inductive types. Indeed, in all of them the return types of constructors are always the same. In some way, they do not exploit the real possibilities of dependent types. What if we wanted constructors to specify that they inhabit a type at some specific value? This is exactly the point of *indexed inductive types*.

The paradigmatic example here is (propositional) equality, an inductive meant to represent equality *internally* to the logic, *i. e.* as a notion on which one can reason – for instance, do proofs by induction –, rather than an external one such as conversion. Rules for equality are given in Figure 3.14. Rule EqType does not depart much from what we have already seen, apart from the fact that it takes not only a type as a parameter, but also a term. Rule EqRefl is already more interesting. Here we can see that the second argument of type A is fixed to be a by the constructor. This gets more visible in Rule EqInd: in order for the branch b to be typeable, the predicate needs to be abstracted not only on the scrutinee, but also on that second argument. Such arguments to an inductive type, whose value depends on the constructor and need to be abstracted over in branches, are called *indices*. By contrast, the other arguments that behave uniformly are called parameters.

As for the logical interpretation, in the simplified case where P only depends on the index, Rule EqInd corresponds to the idea that equal terms should be indiscernible: whenever both $a =_A a'$ and $P[y \coloneqq a]$ hold, then so does $P[y \coloneqq a']$. In words, every property true of a is also true of a'. Paired with the power of dependent types this presentation of equality gives rise to a very rich theory, and forms the basis for the whole line of research in Homotopy Type Theory [Uni13].

[Uni13]: Univalent Foundation Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*

However, in the context of bare CIC, this richness is also a curse, and indexed inductive types can be very tricky to handle. In particular, the work of Part 'Bidirectional Elaboration for Gradual Typing' does not extend well to generic indexed inductive types. There is, however, a somewhat simpler kind of indexed inductive types, where the indices are not any term of any arbitrary type – as in the case of equality –, but inhabitants of an inductive type. Such a case is easier to handle, and is often sufficient, especially for dependently-typed programming. The prototypical example here is that of

$$\textsc{VectType}\ \frac{\vdash A : \square_i \qquad \vdash n : \mathbf{N}}{\Gamma \vdash \mathbf{Ve}\,(A, n) : \square_i} \qquad \textsc{Vnil}\ \frac{\Gamma \vdash A : \square}{\Gamma \vdash \varepsilon_A : \mathbf{Ve}\,(A, 0)}$$

$$\textsc{Vcons}\ \frac{\Gamma \vdash A : \square \qquad \Gamma \vdash n : \mathbf{N} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash l : \mathbf{Ve}\,(A, n)}{\Gamma \vdash a ;;_{A,n} l : \mathbf{Ve}\,(A, \mathrm{S}(n))}$$

$$\textsc{VectInd}\ \frac{\begin{array}{c}\Gamma \vdash s : \mathbf{Ve}\,(A, n) \qquad \Gamma, y : \mathbf{N}, z : \mathbf{Ve}\,(A, y) \vdash P : \square \qquad \Gamma \vdash b_\varepsilon : P[y := 0, z := \varepsilon] \\ \Gamma, y_1 : \mathbf{N}, y_2 : A, y_3 : \mathbf{Ve}\,(A, y_1), p_{y_3} : P[y := y_1, z := y_3] \vdash b_{;;} : P[y := \mathrm{S}(y_1), z := y_2 ;;_{A,y_1} y_3]\end{array}}{\Gamma \vdash \mathrm{ind}_{\mathbf{Ve}}\,(s; y.z.P; b_\varepsilon, y_1.y_2.y_3.p_{y_3}.b_{;;}) : P[y := n, z := s]}$$

$$\iota\textsc{Vnil}\ \frac{}{\mathrm{ind}_{\mathbf{Ve}}\,(\varepsilon_A; y.z.P; b_\varepsilon, y_1.y_2.y_3.p_{y_3}.b_{;;}) \rightharpoonup b_\varepsilon}$$

$$\iota\textsc{Vcons}\ \frac{}{\begin{array}{c}\mathrm{ind}_{\mathbf{Ve}}\,(a ;;_{A,n} l; y.z.P; b_\varepsilon, y_1.y_2.y_3.p_{y_3}.b_{;;}) \rightharpoonup \\ b_{;;}\,[y_1 := n, y_2 := a, y_3 := l, p_{y_3} := \mathrm{ind}_{\mathbf{N}}(l; z.P; b_\varepsilon, y_1.y_2.y_3.p_{y_3}.b_{;;})]\end{array}}$$

Figure 3.15. Vector type

vectors, which we have already encountered in Chapter 2, and is described in detail in Figure 3.15. They are similar to lists, but with a natural number index which records the length of the vector in its type. This allows for finely-grained specification, for instance a head function that takes as input a vector of length at least one, and is thus ensured to never fail on an empty vector by mere virtue of typing.

3.5.5. The Calculus of Constructions

So far we only gave examples of the inductive types one could wish for. A description of how to generally define inductive types and construct induction principles in a way that keeps the good properties of the system would not very enlightening at this point. Let us simply say that the main restriction – barring typing constraints – is to ensure, through a criterion called (strict) positivity, that the recursive structure of the inductive type is well-founded, so that positing its existence does not endanger normalization or consistency.

On paper, rather than a difficult to read general presentation we reuse the previous set of examples to show how our setting adapts to inductive types in their three main complexities – recursion, parameters and indices. But the formalization in MetaCoq handles the general case, in the even more complex setting of PCUIC as presented in Section 3.6.

In the end, when we talk about CIC we mean the extension of CC_ω with any number of inductive types, valid in the previous sense. As already explained, in Part 'Bidirectional Elaboration for Gradual Typing' we need to restrict to non-indexed inductive types. In that setting, our base system is *CIC_*, the restriction of CIC to exclude indexed inductive types.

3.6. Beyond CIC: PCUIC

CIC as described in the previous section is already very expressive and powerful. It is nevertheless still far from a "real-world" type theory such as that implemented in Coq and formalized in MetaCoq, the *Polymorphic, Cumulative Calculus of Inductive Constructions* (*PCUIC*), which extends CIC with many features which are crucial for usability. As some additions of PCUIC are discussed throughout this thesis, we wish to already give a high level idea of them, while reserving the technical details for Part 'A Certified Kernel for Coq, in Coq'.

3.6.1. Cumulativity

The first addition of PCUIC is cumulativity, which allows some extra flexibility with universe levels. To see why this is useful, consider the polymorphic identity function $\lambda(A{:}\,\square_i)(x{:}\,A).\ x$, of type $\Pi\, A{:}\,\square_i.\ A \to A$. If we want to use it at type $\mathbf{N}$, we must force i to be 0. But this means that we cannot use it later on at type $\square_0$! In a concrete system, where a huge number of universe levels appear under the hood, this would quickly become unhandy.

Instead, *cumulativity* – written $\preceq$ – is an extension of conversion with a limited form of subtyping, generated by the inclusion of a universe $\square_i$ in any larger universe $\square_j$. This means that while $\square_i \cong \square_j$ is true only if $i = j$, cumulativity allows for $\square_i \preceq \square_j$ as soon as $i \leq j$. This subtyping can be extended to function types, by allowing $\Pi\, x{:}\,A.\ B \preceq \Pi\, x{:}\,A'.\ B$ whenever $A \cong A'$ and $B \preceq B'$. Note that contrarily to other forms of subtyping, this does not allow for contravariant subtyping on the domain – that would correspond to $A' \preceq A$ –, only for equivariant one – the domains should be convertible. This is because cumulativity is usually modelled using set inclusion [LW11], which straightforwardly handles equivariant subtyping, but not so easily contravariant subtyping.

[LW11]: Lee et al. (2011), *Proof-irrelevant model of CC with predicative induction and judgmental equality*

$$\textsc{UnivCum}\ \frac{i \leq j}{\square_i \preceq \square_j} \qquad \Pi\textsc{Cum}\ \frac{A \cong A' \qquad B \preceq B'}{\Pi\, x{:}\,A.\ B \preceq \Pi\, x{:}\,A'.\ B'}$$

$$\textsc{ConvCum}\ \frac{A \cong A'}{A \preceq A'} \qquad \textsc{Refl}\ \frac{}{A \preceq A} \qquad \textsc{Trans}\ \frac{A \preceq A' \qquad A' \preceq A''}{A \preceq A''}$$

$$\textsc{UCum}\ \frac{\Gamma \vdash T' : \square \qquad T \preceq T'}{\Gamma \vdash T \preceq T' : \square} \qquad \textsc{Cum}\ \frac{\Gamma \vdash t : T \qquad \Gamma \vdash T \preceq T' : \square}{\Gamma \vdash t : T'}$$

Figure 3.16. Rules for declarative cumulativity

To adapt the definitions of declarative conversion to cumulativity, the three important rules are given in Figure 3.16. The first two rules are the ones we already hinted at: Rule UnivCum is the base case for cumulativity, and Rule ΠCum is the relaxed congruence rule for Π-types. The next one, Rule ConvCum, allows to turn any proof of conversion in a cumulativity one, effectively describing how cumulativity behaves outside the fragment formed by Π-types and universes. Next come Rules Refl and Trans, which assert that cumulativity is a pre-order. Of course there is no rule for symmetry, because we do not want cumulativity to be an equivalence relation. Finally,

Rules UCum and Cum show how cumulativity is used: it simply replaces conversion.

As for algorithmic conversion, the important modification is to replace α-equality with an *α-pre-order* $\leq_\alpha$, which extends the former with a rule corresponding to Rule UnivCum: $t \leq_\alpha t'$ means that t and t' have the exact same structure, up to variable names and universe levels, that might be lower in t compared to t'.

3.6.2. The sort of propositions

A second addition in PCUIC, and one that has been a distinctive feature of Coq for a very long time – it is already present in Coquand and Huet [CH88] – is the sort Prop. This is a universe, like $\square_i$, but it is designed to be a type for propositions – hence the name. It has two main distinctive characteristics.

[CH88]: Coquand et al. (1988), *The calculus of constructions*

$$\textsc{Prop}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathrm{Prop} : \square_0}$$

$$\Pi\textsc{TyProp}\ \frac{\Gamma \vdash A : \square \qquad \Gamma, x : A \vdash P : \mathrm{Prop}}{\Gamma \vdash \Pi\, x : A.\ P : \mathrm{Prop}}$$

$$\Pi\textsc{PropProp}\ \frac{\Gamma \vdash A : \mathrm{Prop} \qquad \Gamma, x : A \vdash P : \mathrm{Prop}}{\Gamma \vdash \Pi\, x : A.\ P : \mathrm{Prop}}$$

Figure 3.17. Typing rules for propositions

The first one is its *impredicativity*, meaning that while Prop is at the bottom of the universe hierarchy (Rule Prop), any quantification with a proposition as codomain is again a proposition (Rules ΠTyProp and ΠPropProp). This means that propositions are able to formalize properties of types at any level. Due to this impredicative nature, having such a sort of propositions makes the system much more powerful as a logic, which also makes it much harder to build models of it. Indeed, those usually prove consistency of the modelled system, something which requires having an even higher logical strength than it. Since parts of this thesis – especially Part 'Bidirectional Elaboration for Gradual Typing' – use such models that do not scale to an impredicative sort of proposition, we refrain from including one in our standard CIC. In the other cases, including a sort of propositions makes the system more complex but without raising new interesting questions.

The second defining characteristic of Prop is *proof irrelevance*. This means that PCUIC has a criterion, called singleton elimination, which maintains a form of segregation between terms inhabiting types in $\square$ and those inhabiting types in Prop, ensuring that terms of the first kind cannot depend in a relevant way on terms of the second. For instance, if $P : \mathrm{Prop}$, it ensures that it is impossible to build a function $f : P \to \mathbf{B}$ and two terms $p_1 : P$ and $p_2 : P$ such that $f\ p_1 \cong \mathrm{ff}$ and $f\ p_2 \cong \mathrm{tt}$. This segregation aims at allowing separation between the part of the system that should be seen as programs and that which should be seen as proofs, so that it is possible to write programs decorated with complex correctness proofs, while later on erasing all the logical content to keep only the computational content of the program. This is the erasure procedure we introduced in Section 1.3.2, for which Prop is crucial.

3.6.3. Local definitions

It is often useful to locally introduce a shorthand to be used repeatedly, and this is what PCUIC allows with a new term former, *local definitions* $\mathrm{let}\ x : A := t\ \mathrm{in}\ u$. In such a local definition, x can be used in the term u as a shorthand for t.

The main impact of this addition is its effect on contexts: as Rule LetIn illustrates, when typing u, not only the type of the definition is recorded,

$$\textsc{LetIn}\ \frac{\Gamma \vdash A : \square \qquad \Gamma \vdash t : A \qquad \Gamma, x := t : A \vdash u : B}{\Gamma \vdash \text{let } x : A := t \text{ in } u : \text{let } x : A := t \text{ in } B}$$

Figure 3.18a. Typing for local definitions

but also its value t. This is again due to dependency, because the value of the definition, and not only its type, might be needed for u to be well-typed. As an example, suppose we have a function

$$\text{head} : \Pi(A : \square)(x : \mathbf{N}).\ \mathbf{Ve}\,(A, \mathrm{S}(x)) \to A$$

and consider

$$\text{let } x : \mathbf{N} := 1 \text{ in } \lambda\, v : \mathbf{Ve}\,(\mathbf{B}, x).\ \text{head}\ \ \mathbf{B}\ 0\ v$$

This term is well-typed only if the fact that x has value 1 is available in the right-hand side.

This also means that contexts now should be recorded in conversion and cumulativity, because those need to access the value of a variable bound by a definition if we want to enable the behaviour just described. In the end, there are two top-level reductions for definitions, given opposite: they can be either simplified right away into a substitution (ζRed)[17], or recorded into the context and simplified only later on using Rule δRed.

$$\zeta\textsc{Red}\ \frac{}{\Gamma \vdash \text{let } x : A := t \text{ in } u \rightharpoonup u[x := t]}$$

$$\delta\textsc{Red}\ \frac{(x := t : A) \in \Gamma}{\Gamma \vdash x \rightharpoonup t}$$

Figure 3.18b. Top-level reduction for local definitions

17: The notations are a bit misleading here: the local definition is part of the syntax of terms, while substitution is a meta-level operation. While the former encodes the latter in the syntax, they are quite different!

3.6.4. Global environments

PCUIC offers a second way to record definitions, inside a so-called *global environment*. The difference between this and the addition of local definitions in a context we just saw is motivated by rather concrete considerations. The (local) context corresponds to definitions and abstractions encountered when type-checking a single proof or program, and should thus be relatively shallow – the order of magnitude is a dozen variables – but it might change very often, with variable being both added and abstracted over. The environment, on the contrary, can become huge – corresponding to a whole library with thousands of components — but changes less often, and usually in a monotone way – new definitions are added, but not removed. Therefore, typing in PCUIC actually has an extra parameter: it is of the form $\Sigma;;;\Gamma \vdash t : T$, with Σ corresponding to the environment.

This environment is not only used for definitions and assumptions, but also to keep track of inductive types. It thus effectively implements our somewhat vague assumption that CIC is extended with "any number of valid inductive types". Of course, there is a notion of environment well-formation, which accounts for the fact that it should only contain objects that are well-typed, together with other constraints, for instance that inductive types respect the strict positivity criterion.

There is a further use for this environment: it also records the level variables available for universes, and their constraints. Indeed, in PCUIC, universe levels are expressions rather than simple natural numbers, and the order between expressions is relative to a given environment Σ. There are actually two kinds of those universe variables. The first are global ones, that are recorded in an ever-growing fashion in the environment. This is the older

[Pol92]: Pollack (1992), *Typechecking in Pure Type Systems*

approach, that was introduced in Coq together with typical ambiguity, following Pollack [Pol92].

This approach is however still not flexible enough, which is why a second kind of variables were more recently introduced [ST14]. These are attached locally to an entry in the environment, corresponding to a form of *universe polymorphism*, and each time such a definition is used it can be instantiated with new universe levels.[18] This is for instance useful to have a single (polymorphic) definition of categories, and still be able to define the category – at level j – of all categories – at a level $i < j$ –, by instantiating the definition at the two different levels i and j. If there was just one global level k, then doing this would result in a constraint $k < k$, and this definition would not be accepted.

[ST14]: Sozeau et al. (2014), *Universe Polymorphism in Coq*

18: This is somewhat similar to the Hindley-Milner style of type polymorphism [Hin69; Mil78] widely used in the ML family of languages, albeit with universe levels rather than types.

[Hin69]: Hindley (1969), *The Principal Type-Scheme of an Object in Combinatory Logic*

[Mil78]: Milner (1978), *A theory of type polymorphism in programming*

3.6.5. Enhanced inductive types

Of course inductive types in PCUIC are also affected by these extensions. Not only can they be polymorphic, as definitions, they also feature a form of cumulativity, that makes this polymorphism more seamless – see Timany and Sozeau [TS18] for a precise description. This for instance prevents issues with ε_A not being of type $\mathbf{Li}\,(A)$ because of a mismatch between type variables – those do not appear in our presentation of CIC, but are present in PCUIC due to the general setting for polymorphic inductive types.

[TS18]: Timany et al. (2018), *Cumulative Inductive Types In Coq*

Moreover, the strict positivity criterion adopted in PCUIC is very general, as it allows mutually defined and nested inductive types. The former are multiple inductive types defined at the same time, where a constructor of one type can take a recursive argument of another. For instance, an inductive oddness/evenness predicate with constructors $\mathrm{oddS} : \Pi\, x{:}\,\mathbf{N}.\ \mathrm{even}\, x \to \mathrm{odd}\, \mathrm{S}(x)$ and $\mathrm{evenS} : \Pi\, x{:}\,\mathbf{N}.\ \mathrm{odd}\, x \to \mathrm{even}\, \mathrm{S}(x)$. The latter are types where a constructor can take a recursive argument mentioning the type being defined as a parameter to another inductive type – for instance, a type of tree where a node takes a list of trees as argument.

But the most significant difference is that the induction principles, such as the ones we gave for CIC, are replaced with two new constructions: pattern-matching and fixed-points. The first corresponds to the non-recursive component of the induction principles, while the second allows to define a function that calls itself recursively. To avoid paradoxical definitions, not every recursive definition is accepted, however. Instead, there is a restriction called the *guard condition* to how a recursive function can be defined, which amounts to checking that recursive calls are made on structurally smaller sub-terms – by means of pattern-matching. This guard condition theoretically ensures that fixed-points and pattern-matching can always be reduced to recursors [Gim95], which are what proofs of normalization and/or consistency usually consider. However, in practice the former is much more flexible and natural to use than the latter.

[Gim95]: Giménez (1995), *Codifying guarded definitions with recursive schemes*

3.6.6. Records and co-inductive types

The last ingredient in PCUIC goes beyond inductive types, by adding more primitive types to the theory.

The first kind are *record types*, a generalization of Σ-types which allows for any number of named fields. The main addition of record types is the ability to access those fields via *projections* rather than by using pattern-matching. For the Σ-type as presented in Section 3.5, this would mean accessing the two fields of the pair p with two term formers $p.1$ and $p.2$. These record types are very useful to package objects together, be it in programming or in mathematics – where such bundles are ubiquitous, for instance when formalizing hierarchies of mathematical structures [CST20].

[CST20]: Cohen et al. (2020), *Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi*

The second kind are *co-inductive types*. These are somewhat similar to inductive types, but while the latter correspond to well-founded objects, the former represent potentially infinite objects, such as streams of values. Because of this flavour of infinity, co-inductive types pose an inherent threat to good properties of the system, in particular decidability of type-checking. At the time of their introduction in Coq [Gim95], they were presented in a so-called "positive" fashion – close to the presentation of inductive types –, which kept normalization at the cost of subject reduction. Another presentation, inspired by more recent work on co-induction, and especially copatterns [Abe+13], is the "negative" one – similar to the projection-based presentation of record types –, which regains the good properties of the system. While the older positive presentation is still present in Coq, in part for compatibility reasons, only the negative one is formalized in MetaCoq.

[Gim95]: Giménez (1995), *Codifying guarded definitions with recursive schemes*

[Abe+13]: Abel et al. (2013), *Copatterns: programming infinite structures by observations*

Bidirectional Calculus of Inductive Constructions

When presenting a typing derivation the way we did in Chapter 3, there is an important piece of information missing. In logical programming, this is called the mode of the inference rules, *i.e.* which objects are considered as inputs and which as outputs in the search for a derivation. This information, however, is crucial when one tries to build a type-checker: some rules might seem fine when writing them down on paper, but trying to give them a sensible mode fails, indicating they are not suited for an implementation. In the case of the typing judgement $\Gamma \vdash t : T$, usually both the term t under inspection and the context Γ are inputs – although some depart from this Jim [Jim96]. The mode of the type T, however, is much less clear: should it be inferred based upon Γ and t, or do we merely want to check whether t conforms to a given T? Both are sensible questions, and in fact typing algorithms for complex type systems usually alternate between them during the inspection of a single term/program. The bidirectional approach makes this difference between modes explicit, by decomposing *undirected typing*[1] $\Gamma \vdash t : T$ into two separate judgements $\Gamma \vdash t \triangleright T$ (inference) and $\Gamma \vdash t \triangleleft T$ (checking)[2], that differ only by their modes. The type is an input in inference, but an output in checking. Following this decomposition[3] leads to type systems that are more structured and directly amenable to implementations, and to good quality algorithms.[4]

This is appealing, but in the dependently typed world, despite advocacy by *e.g.* McBride [McB18; McB19] to adopt this approach during the design of type systems on paper rather than in their implementations only, most of the theoretical work to this day remains undirected. Bidirectionality appears mostly in articles concentrating on the description of typing algorithms, for instance Huet [Hue89], Coquand [Coq96], or Norell [Nor07]. However, since these primarily insist on the algorithmic aspect, they do not consider the bidirectional structure for itself. Moreover, in the case of Coquand [Coq96] and Norell [Nor07], they concentrate on bidirectional typing as a way to remedy for the lack of annotations on their Curry-style λ-abstractions. This is sensible when looking for lightness of the input syntax, but poses an inherent completeness problem: a term such as $(\lambda\, x.x)\ 0$ is not typeable in those systems.[5] In the context of Church-style abstraction, the closest there is to a description of bidirectional typing for CIC is probably the one given by the Matita team [Asp+12], which however concentrates again on the challenges posed by the elaboration and unification algorithms. They also do not consider the problem of completeness with respect to a given undirected system, as it would fail in their setting due to the undecidability of higher order unification.

In this part (Bidirectional Calculus of Inductive Constructions), we wish to fill this gap in the literature, by describing a bidirectional type system that is complete with respect to (undirected) CIC, as presented in Chapter 3. By completeness, we mean that any term that is typeable in the undirected system should also infer a type in the bidirectional one. This feature is very desirable when implementing kernels for proof assistants, whose algorithms should correspond to their undirected specification – even on terms not in normal form. Indeed, reduction is only normalizing on well-typed terms, so it should not be called on a term that is not known to be well-typed. Thus if a developer wishes to generate a term using tactics, they cannot use reduction before knowing that it is well-typed, but might not be able to type-check it because it is not a normal form... And ensuring that a tactic returns normal forms only might be unfeasable, and should not be a

[Jim96]: Jim (1996), *What Are Principal Typings and What Are They Good For?*

1: We call anything related to the $\Gamma \vdash t : T$ judgement undirected, by contrast with bidirectional typing.

2: We use the $\triangleright$ and $\triangleleft$ symbols rather than the more usual $\Rightarrow$ and $\Leftarrow$ to avoid confusion with implication and with the Coq notation for functions.

3: Pioneered by Pierce and Turner [PT00], a general survey can be found in Dunfield and Krishnaswami [DK21].

[PT00]: Pierce et al. (2000), *Local Type Inference*

[DK21]: Dunfield et al. (2021), *Bidirectional Typing*

4: Pierce and Turner [PT00] for instance stress good error reporting as an important property of their approach.

[McB18]: McBride (2018), *Basics of Bidirectionalism*

[McB19]: McBride (2019), *Check the Box!*

[Hue89]: Huet (1989), *The Constructive Engine*

[Coq96]: Coquand (1996), *An algorithm for type-checking dependent types*

[Nor07]: Norell (2007), *Towards a practical programming language based on dependent type theory*

5: They are actually only able to give types to normal forms.

[Asp+12]: Asperti et al. (2012), *A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions*

concern.

The bidirectional system we present naturally forms an intermediate step between actual algorithms and undirected type systems, something we exploit in Part 'A Certified Kernel for Coq, in Coq'. But its interest is not limited to the relation with implementations. Indeed, the structure of a bidirectional derivation is more constrained than that of an undirected one, especially controlling the usage of computation – *e.g.* the conversion rule Referencesrule:cic-conv. This finer structure can make proofs easier, while the equivalence ensures that they can be transported to the undirected world. We show this by providing straightforward proofs of uniqueness of types up to cumulativity, and of strengthening.

As we did in Chapter 3, we start by exposing the main ideas in the simpler setting of CC_ω, in Chapter 4. With those set clear, we go on with their adaptation to PCUIC, and the subtle issues that arose in that context, in Chapter 5. Finally, Chapter 6 describes early investigations into giving a bidirectional treatment not only of typing, but also of conversion.

4. Warm-up: CCω

4.1 Turning CC_ω Bidirectional . 51
4.1.1 McBride's discipline 51
4.1.2 The typing rules 53
4.1.3 Constrained inference in disguise 54
4.2 Properties of the Bidirectional System 55
4.2.1 Correctness 55
4.2.2 Completeness 56
4.2.3 Uniqueness 58
4.2.4 Strengthening 59

4.1. Turning CC_ω Bidirectional

4.1.1. McBride's discipline

To design our bidirectional type system, we follow a discipline exposed by McBride [McB18; McB19]. The central point is to distinguish in a judgement between three *modes*: the *subject*, whose well-formation is under examination, *inputs*, whose well-formation is a condition for the judgement to be meaningful, and *outputs*, whose well-formation should be a consequence of the judgement. By *well-formed*, which we use indistinctly for contexts, terms and types, we mean:

- $\vdash \Gamma$ in the case of a context Γ,
- $\Gamma \vdash T : \square$ in the case of a type T,
- the existence of some T such that $\Gamma \vdash t : T$ in the case of a term t.

For the last two, this is relative to an implicit context Γ. We also use *well-typed* for a term, with the same meaning as well-formed.

In the case of *inference* $\Gamma \vdash t \triangleright T$, the subject is t, Γ is an input and T is an output. On the contrary, in *checking* $\Gamma \vdash t \triangleleft T$, t is still the subject and Γ is an input, but this time T is an input as well. This means that one should consider whether $\Gamma \vdash t \triangleright T$ only in cases where $\vdash \Gamma$ is already known, and if the judgement is derivable it should be possible to conclude that not only t, but also T are well-formed.

In order to enforce this property globally, all inference rules should locally preserve it as an invariant.[1] More precisely, information flows in a clockwise manner. First, we can assume that inputs to the conclusion are well-formed, as inputs to the whole rule. Next, we move to the premises. Here the constraint is reversed: we should ensure that inputs to a premise are well-formed, but can assume that its outputs and subjects are. We might need to use the well-formation of subjects or outputs of previous ones for that. Finally, information goes to the conclusion again, and now not only the subject but also the output should be well-formed if all those of the premises are.

1: The motto – slightly adapted from McBride [McB18] – is: *A rule is a server for its conclusion and a client for its premises. Servers receive promises about inputs and make promises about outputs, clients make promises about inputs and receive promises about outputs.*

[McB18]: McBride (2018), *Basics of Bidirectionalism*
[McB19]: McBride (2019), *Check the Box!*

(1)
$$\frac{\Gamma \vdash t \triangleright_\Pi \Pi x{:}A.B \qquad \Gamma \vdash u \triangleleft A}{\Gamma \vdash t\ u \triangleright B[x := u]}$$

(2)
$$\frac{\Gamma \vdash t \triangleright_\Pi \Pi x{:}A.B \qquad \Gamma \vdash u \triangleleft A}{\Gamma \vdash t\ u \triangleright B[x := u]}$$

(3)
$$\frac{\Gamma \vdash t \triangleright_\Pi \Pi x{:}A.B \qquad \Gamma \vdash u \triangleleft A}{\Gamma \vdash t\ u \triangleright B[x := u]}$$

(4)
$$\frac{\Gamma \vdash t \triangleright_\Pi \Pi x{:}A.B \qquad \Gamma \vdash u \triangleleft A}{\Gamma \vdash t\ u \triangleright B[x := u]}$$

(5)
$$\frac{\Gamma \vdash t \triangleright_\Pi \Pi x{:}A.B \qquad \Gamma \vdash u \triangleleft A}{\Gamma \vdash t\ u \triangleright B[x := u]}$$

Figure 4.1. An illustration of McBride's discipline (well-formed objects are in blue, those not known to be so are in red)

As an illustration, an example of a rule that respects this discipline – that for application – is given in Figure 4.1. Let us ignore for an instant the exact meaning of the judgement $\triangleright_\Pi$ which we introduce soon, and whose modes are the same as inference. Instead, focus on well-formation: objects known to be well-formed are on a blue background, those which are not are on a red one. First, Γ is well-formed, as an input to the conclusion (1). Thus we can thus move to the first premise, since its only input is Γ. From that premise holding, we learn that t and $\Pi\, x{:}\, A.\ B$ are well-formed (2). Therefore, A is in particular well-formed, and we can move to the second premise whose two inputs are now known to be well-formed (3). From it, we learn that u is well-formed (4). Now we can deduce that $t\ u$ is well-formed. But this is not enough: since $B[x \coloneqq u]$ is an output of the conclusion, we must ensure it is well-formed too. Fortunately, it is, since $\Pi\, x{:}\, A.\ B$ is, so B is too,[2] and so $B[x \coloneqq u]$ is as well, since $\Gamma \vdash u \triangleleft A$. Thus, we can move back to the conclusion (5), which ends our roundtrip.

2: In the extended context $\Gamma, x{:}\, A$.

A somewhat similar discipline has appeared independently in Dunfield and Krishnaswami [DK21, Section 4], where it is called "Pfenning's recipe". The main criterion is *mode-correctness*, which demands an information flow similar to McBride's, but is coarser, as it does not consider well-formation of the objects, only their knowledge. For instance, in the case of $\lambda\, x{:}\, A.\ t$, that criterion allows to directly extend a context with A to infer a type for t, because it is known, but McBride's discipline forbids it, because A's well-formation is not established. Another related condition is also used in Bauer, Haselwarter, and Lumsdaine [BHL20]. The authors introduce the notions of a *(weakly) presuppositive type theory* [BHL20, Def. 5.6] and of *well-presented* premise-family and rule-boundary [BHL20, Def. 6.16 and 6.17], using what they call the *boundary* of a judgement as the analogue of our inputs and outputs. Due to their setting being undirected, this is however more restrictive, because they are not able to distinguish inputs from outputs and thus cannot relax the condition to only demand inputs to be well-formed but not outputs.

[DK21]: Dunfield et al. (2021), *Bidirectional Typing*

[BHL20]: Bauer et al. (2020), *A general definition of dependent type theories*

Because of our dependently typed setting, we actually need to introduce a third judgement, beyond the already mentioned inference and checking: *constrained inference*, written $\Gamma \vdash t \triangleright_h T$, where h is either Π or $\square$.[3] Constrained inference is a judgement[4] with the exact same modes as inference, but where the type output is not completely free. Rather, as the name suggests, a constraint is imposed on it, namely that its head constructor can only be the corresponding element of h. This is needed to handle the behaviour absent in simple types that some terms might not have a desired type "on the nose". Take for instance the first premise $\Gamma \vdash t : \Pi\, x{:}\, A.\ B$ of Rule App. What bidirectional judgement should replace it? It would be too much to ask t to directly infer a Π-type, as some reduction might be needed to uncover this Π. Checking also cannot be used, because the domain and codomain of the tentative Π-type are not known at that point: they should be inferred from t.

3: These are the only type formers in CC_ω, but in PCUIC, h can also be *e.g.* an inductive type.

4: Or, rather, a family of judgements indexed by h.

Finally, this mode distinction also applies to computation-related judgements, although those have no subject. Instead, what is under scrutiny is the "computational content" of the rule. For conversion $\Gamma \vdash T \cong T' : \square$, both T and T' are inputs and thus should be known to be well-formed beforehand. For reduction $T \to^\star T'$, on the contrary, T is an input, but T' is an output. Hence, only T needs to be well-formed *a priori*, and we rely on the subject reduction property to ensure that the output T' also is.

4.1.2. The typing rules

To transform the rules of CC_ω as given in Figure 3.2, start by recalling that we wish to obtain a complete bidirectional type system. Therefore, any term should infer a type, and thus all rules where the subject of the conclusion starts with a term former should give rise to a rule with inference as a conclusion. It thus remains to choose the judgements for the premises, which amounts to determining their modes. If a term in a premise appears as input in the conclusion or output of a previous premise, then it can be considered an input, otherwise it must be an output. Moreover, if a type output is unconstrained, then inference can be used, otherwise we must resort to constrained inference. This transformation leads to the rules of Figure 4.2a.

$$\textsc{Var}\ \frac{(x:T) \in \Gamma}{\Gamma \vdash x \triangleright T} \qquad \textsc{Univ}\ \frac{}{\Gamma \vdash \square_i \triangleright \square_{i+1}}$$

$$\Pi\textsc{Ty}\ \frac{\Gamma \vdash A \triangleright_\square \square_i \qquad \Gamma, x:A \vdash B \triangleright_\square \square_j}{\Gamma \vdash \Pi\, x:A.\, B \triangleright \square_{\max(i,j)}}$$

$$\textsc{Abs}\ \frac{\Gamma \vdash A \triangleright_\square \square_i \qquad \Gamma, x:A \vdash t \triangleright B}{\Gamma \vdash \lambda\, x:A.\, t \triangleright \Pi\, x:A.\, B}$$

$$\textsc{App}\ \frac{\Gamma \vdash t \triangleright_\Pi \Pi\, x:A.\, B \qquad \Gamma \vdash u \triangleleft A}{\Gamma \vdash t\ u \triangleright B[x := u]}$$

Figure 4.2a. Rules for inference in bidirectional CC_ω

In anticipation, we set the typing rules for CC_ω so that this transformation would be direct. This particularly applies to the undirected Rule Abs, recalled opposite. Indeed, there are at least two other ways to write it, which do not lead to a valid bidirectional presentation. The first, which is the usual one in Pure Type Systems (PTS) [Bar91], is to have $\Gamma \vdash \Pi\, x:A.\, B : \square$ as a premise instead of $\Gamma \vdash A : \square$. In the setting of a general PTS, this is needed, because not every Π-type is well-formed, even if the domain and codomain are.[5] However, this premise is problematic in the bidirectional setting. Indeed, B can only be inferred as a type for the body of the abstraction t. But to infer a type for t, the context $\Gamma, x:A$ needs to be well-formed, which is not known if this premise is the first one. This issue has been identified by Pollack [Pol92], who remarked that the bidirectional structure we present here is only equivalent to the undirected one in semi-full PTS – a slight generalization of the full ones. In a full PTS, the opposite path of simply removing the first premise altogether can also be taken, relying on validity to ensure that $\vdash \Gamma, x:A$ and thus $\Gamma \vdash A : \square$. But again, in a bidirectional setting, this does not respect McBride's discipline.

$$\textsc{Abs}\ \frac{\Gamma \vdash A : \square \qquad \Gamma, x:A \vdash t : B}{\Gamma \vdash \lambda\, x:A.\, t : \Pi\, x:A.\, B}$$

[Bar91]: Barendregt (1991), *An Introduction to Generalized Type Systems*

5: PTS where this is true are called *full*.

[Pol92]: Pollack (1992), *Typechecking in Pure Type Systems*

The main difference between the bidirectional and undirected rules is that we dropped hypotheses of context well-formation in Rules Univ and Var. Indeed, since the context is always supposed to be well-formed as an input to the conclusion, it is not useful to re-check it. This is also in line with implementations, where the context is not checked at leaves of a derivation tree, with performance issues in mind. The well-formation invariants then ensure that any derivation starting with the (well-formed) empty context will only ever encounter well-formed contexts.

$$\textsc{Check}\ \frac{\Gamma \vdash t \triangleright T' \qquad T' \cong T}{\Gamma \vdash t \triangleleft T}$$

$$\textsc{UnivInf}\ \frac{\Gamma \vdash t \triangleright T \qquad T \to^\star \square_i}{\Gamma \vdash t \triangleright_\square \square_i}$$

$$\Pi\textsc{Inf}\ \frac{\Gamma \vdash t \triangleright T \qquad T \to^\star \Pi\, x{:}\, A.B}{\Gamma \vdash t \triangleright_\Pi \Pi\, x{:}\, A.B}$$

Figure 4.2b. Computation rules for bidirectional CC_ω

With the rules for term formers taken care of, we are left with the single Rule Conv. There are two different possible adaptations of this rule, depending on modes for computation. In the case of checking, the target type is an input, so it can be compared to the inferred one using conversion. But in the case of constrained inference it is unknown, and so we must resort to reduction to obtain it from the inferred one. Using conversion would not respect modes, since it has two inputs. This eventually leads to the decomposition of Rule Conv into Rule Check in the first case, while UnivInf and ΠInf correspond to the second case. Note that while the way conversion and reduction can be used in derivations have changed, those relations themselves remain untouched, we only refined them by giving them an explicit mode. We also do not need to choose one or the other notion of conversion yet. Instead, we can stay abstract, only listing the properties we need from it in order to establish the equivalence.

4.1.3. Constrained inference in disguise

This need to split the conversion rule into a reduction and conversion subroutines depending on the mode is of course known to the implementors of proof assistants [AA11]. It explains in part the ubiquity of weak-head reduction in the dependently typed setting. Indeed, it is exactly the minimal reduction strategy that is needed to expose the head constructor of a type, and thus to implement constrained inference.

[AA11]: Abel et al. (2011), *A Partial Type Checking Algorithm for Type:Type*

Still, reduction is only a means to determine whether a certain term fits into a certain kind of types. In the setting of CC_ω, this is basically the only way to do. However, as soon as conversion is extended or modified, reduction is often not enough any more. Putting constrained inference forward explains some ideas that recurrently appear in such settings: they are not ad-hoc workarounds, but are based on the need to account for constrained inference.

[Pol92]: Pollack (1992), *Typechecking in Pure Type Systems*

We already mentioned Pollack [Pol92], where $\Gamma \vdash t : T$ is used for inference, and a judgement written $\Gamma \vdash t :\geq T$ – denoting type inference followed by reduction – is used to effectively inline the two hypothesis of our constrained inference rules. Checking is also inlined. Similarly, Abel, Coquand, and Dybjer [ACD08] use a judgement written $\Delta \vdash V\delta \Uparrow \mathrm{Set} \rightsquigarrow i$, where a type V is checked to be well-formed, but with its exact level i free. This corresponds very closely to our use of $\triangleright_\square$. Gratzer, Sterling, and Birkedal [GSB19] similarly use a judgement $\Xi \vdash T \Leftarrow type$, but they do not bother inferring the level as they never have any need for it.

[ACD08]: Abel et al. (2008), *Verifying a Semantic βη-Conversion Test for Martin-Löf Type Theory*

[GSB19]: Gratzer et al. (2019), *Implementing a Modal Dependent Type Theory*

But the main area where constrained inference repeatedly becomes apparent is that of elaboration. For instance, Saïbi [Saï97] describes an elaboration mechanism inserting coercions between types. This happens primarily during checking, when both types are known. However, Saïbi introduces two special classes to handle the need to cast a term to a sort or a function type without more information, exactly in the places where we resort to constrained inference instead of checking. More recently, Sozeau [Soz07] describes a system where conversion is augmented to handle subset types. As in Pollack [Pol92], $\Gamma \vdash t : T$ is used for inference, and the other judgements are inlined. Once again, reduction is not enough to perform constrained inference, this time because type constructors can be hidden in subsets: an inhabitant of a type such as $\{f : \mathbf{N} \to \mathbf{N} \mid f\ 0 = 0\}$ should be usable as

[Saï97]: Saïbi (1997), *Typing Algorithm in Type Theory with Inheritance*

[Soz07]: Sozeau (2007), *Subset Coercions in Coq*

a function of type $\mathbf{N} \to \mathbf{N}$. An erasure procedure is therefore required on top of reduction to remove subsets in the places where we use constrained inference.

Analogous ideas can also be found in Matita's elaboration algorithm, as described in Asperti et al. [Asp+12]. Indeed, the presence of unification meta-variables on top of coercions makes it even clearer that a specific treatment of what we identified as constrained inference is required. In the case of $\triangleright_\Pi$, they have two rules to apply a function, one where its inferred type reduces to a Π-type, corresponding to Rule ΠInf, and another one to handle the case when the inferred type instead reduces to a meta-variable. As Saïbi and Sozeau, they also need to handle coercions for terms in function position. However, their solution is different: they introduce new meta-variables for the domain and codomain, and rely on unification, which is available in their setting, to find values for those. They also need to introduce a special judgement they call *type-level enforcing*, which corresponds to our $\triangleright_\square$ judgement. The solution they take for Π-types is not viable there, as one would need a kind of universe meta-variable. Instead, they rely on backtracking to test multiple possible universe choices.

[Asp+12]: Asperti et al. (2012), *A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions*

Finally, in Part 'Bidirectional Elaboration for Gradual Typing', somewhat akin to the use of meta-variables in Asperti et al. [Asp+12], there are two rules per constrained inference judgement. One when the head constructor is the desired one – as for CC_ω –, and a second one to handle the wildcard ?, characteristic of gradual type systems.

4.2. Properties of the Bidirectional System

Let us now state and sketch proofs of the main properties of the bidirectional system. The first two relate it to the undirected one: it is both correct – terms typeable in the bidirectional system are typeable in the undirected system – and complete – all terms typeable in the undirected system are also typeable in the bidirectional system. Next, we investigate uniqueness of types, and its relation to the choice of a strategy for reduction. Finally, we expose how strengthening can be shown for undirected CC_ω by proving it on the bidirectional side.

4.2.1. Correctness

A bidirectional derivation can be seen as a refinement of an undirected derivation. Indeed, the bidirectional structure can be erased to obtain an undirected derivation, replacing each bidirectional rule with the corresponding undirected rule. As bidirectional rules lack some premises of the undirected ones, missing some sub-derivations must be retrieved by relying on the well-formation invariants going with McBride's discipline. Thus, we get the following correctness theorem – note how the discipline manifests as well-formation hypothesis on inputs.

Theorem 4.1. *Correctness* of bidirectional typing for CC_ω

> If Γ is well-formed and $\Gamma \vdash t \triangleright T$ or $\Gamma \vdash t \triangleright_h T$, then $\Gamma \vdash t : T$. If both Γ and T are well-formed and $\Gamma \vdash t \triangleleft T$, then $\Gamma \vdash t : T$.

Proof.

By mutual induction on the bidirectional typing derivation.
Each rule of the bidirectional system can be replaced by the corresponding rule of the undirected system, with all three rules Check, UnivInf and ΠInf replaced by Conv. In all cases, the induction hypothesis can be used on sub-derivations of the bidirectional judgement, because context extensions and checking are done with types that are known to be well-formed,[6] by induction hypothesis on previous premises and possibly validity.
Some sub-derivations of the undirected rules that have no counterpart in the bidirectional ones are however missing. In Rules Univ and Var, the hypothesis that Γ is well-formed is enough to get the required premise. For Rule Check, the well-formation hypothesis on the type is needed to get the typing premise of UConv. As for Rules UnivInf and ΠInf, that typing premise is obtained by combining the induction hypothesis, validity and subject reduction.
Alternatively, the appeal to validity could be removed by strengthening the theorem to incorporate the well-formation of outputs on top of that of the subject. Here we follow the proofs in MetaCoq, which establishes meta-theoretical properties of the undirected system first – including validity –, so we can exploit these. □

6: This is the point where following McBride's discipline is crucial!

4.2.2. Completeness

Contrarily to correctness, which keeps the structure of a derivation, completeness is of a different nature. Because in bidirectional derivations the computation rules are much less liberal than in undirected ones, the structure of derivations must be altered. The crux of the proof is thus to ensure that all uses of Rule Conv can be permuted down through the other rules, in order to concentrate them in the places where they are authorized in the bidirectional derivation. In a way, composing completeness with correctness gives a kind of normalization procedure on undirected derivations, which produces a canonical one by pushing conversion down as much as possible.

The proof mainly relies on the following lemma, which can be seen as a strong form of injectivity of type constructors – the version of Property 3.9 is a direct consequence.

Lemma 4.2. Conversion implies reduction for type constructors

If $T \cong \square_i$, then $T \to^\star \square_i$.

If $T \cong \Pi\, x{:}\, A.\, B$, then there exist A' and B' such that:

- $T \to^\star \Pi\, x{:}\, A'.\, B'$
- $A' \cong A$
- $B' \cong B$

Proof.

Let us spell out the proof on Π-types – the case of $\square$ is similar, but easier. For algorithmic conversion, by definition there must exist T' and T'' such that $T \to^\star T'$, $\Pi\, x{:}\, A.\ B \to^\star T''$, $T' =_\alpha T''$. But there can be no top-level reduction step in $\Pi\, x{:}\, A.\ B \to^\star T''$, so actually T'' is some $\Pi\, x{:}\, A''.\ B''$ and $A \to^\star A''$, $B \to^\star B''$. Similarly, T' must be some $\Pi\, x{:}\, A'.\ B'$ such that $A' =_\alpha A''$ and $B' =_\alpha B''$. Combining these, we obtain that $A' \cong A$ and $B' \cong B$, as expected.
For declarative conversion, we can go through the equivalence with algorithmic conversion – and thus use confluence under the hood. $\square$

Theorem 4.3. *Completeness* of bidirectional typing for CC_ω

If $\Gamma \vdash t : T$, then there exists T' such that $\Gamma \vdash t \triangleright T'$ and $T' \cong T$.

Proof.

By induction on the undirected typing derivation.

Rules Var and Univ are base cases, and can be simply replaced by the corresponding bidirectional rules. In the case of Rule Conv, the property is a direct consequence of the induction hypothesis, together with transitivity of conversion: we simply conflate two conversions together.

As for Rule ΠTy, the induction hypothesis on the domain A gives the existence of T_A such that $\Gamma \vdash A \triangleright T_A$ and $T_A \cong \square_i$. Using Lemma 4.2, we can derive $\Gamma \vdash A \triangleright_\square \square_i$. Applying a similar reasoning on the codomain and combining both is enough to conclude.

In Rule Abs, we do the same reasoning again on the type annotation. Combined with the induction hypothesis on the body t, we get $\Gamma \vdash \lambda\, x{:}\, A.\ t \triangleright \Pi\, x{:}\, A.\ B'$ for some B' such that $B \cong B'$, and thus $\Pi\, x{:}\, A.\ B \cong \Pi\, x{:}\, A.B'$ as desired.

We are finally left with Rule App. Again, the key is Lemma 4.2, which can be combined with the induction hypothesis on the function f to get $\Gamma \vdash f \triangleright_\Pi \Pi\, x{:}\, A'.\ B'$ for some A' and B' such that $A \cong A'$ and $B \cong B'$, where $\Pi\, x{:}\, A.B$ is the type of f in the undirected derivation. The induction hypothesis on the argument u gives $\Gamma \vdash u \triangleright A''$ with $A'' \cong A$. Thus, by transitivity of conversion $\Gamma \vdash u \triangleleft A'$, and we can apply Rule App to conclude. $\square$

Interestingly, the proof of correctness relies on subject reduction, which itself needs injectivity of type constructors and transitivity of conversion. Similarly, completeness relies both on the injectivity as given by Lemma 4.2, and transitivity of conversion. Be it for algorithmic or declarative conversion, one at least of those is not directly provable – we need confluence. We already hit this same tension between injectivity and transitivity with subject reduction, and must draw the same conclusion: there is no free lunch!

Theorem 4.3 is quite specific to our Church-style design. Instead, an important portion of the research on bidirectional typing in the context of dependent types adopts a Curry-style approach. This is the case of *e.g.* Coquand [Coq96], the type system of Agda as described by Norell [Nor07], and most of the work by Abel [AC07; ACD08; AA11; AÖV17], and McBride [McB16; McB18; McB19]. In such systems, λ-abstractions can only be checked against a given type, but cannot infer one, which implies that only terms with no

[Coq96]: Coquand (1996), *An algorithm for type-checking dependent types*
[Nor07]: Norell (2007), *Towards a practical programming language based on dependent type theory*
[AC07]: Abel et al. (2007), *Untyped Algorithmic Equality for Martin-Löf's Logical Framework with Surjective Pairs*
[ACD08]: Abel et al. (2008), *Verifying a Semantic βη-Conversion Test for Martin-Löf Type Theory*
[AA11]: Abel et al. (2011), *A Partial Type Checking Algorithm for Type:Type*
[AÖV17]: Abel et al. (2017), *Decidability of Conversion for Type Theory in Type Theory*
[McB16]: McBride (2016), *I Got Plenty o' Nuttin'*
[McB18]: McBride (2018), *Basics of Bidirectionalism*
[McB19]: McBride (2019), *Check the Box!*

redexes are typeable. Norell [Nor07] argues that explicit redexes are uncommon in real-life programs, so that being unable to type them is not a strong limitation in practice. Another solution, taken by McBride [McB22], is to add type annotations in order regain the ability to check non-normal terms, at the cost of inserting annotations at the right place. In all cases, however, the fact that all terms well-typed in the declarative system infer a type is irremediably lost. Weaker forms of completeness should still hold for such systems, typically one where all terms check against their type, but are not ensured to infer. See for instance Gratzer, Sterling, and Birkedal [GSB19, Theorem 7.3] for one restricted to normal forms – and thus not taking the role of annotations into account.

[GSB19]: Gratzer et al. (2019), *Implementing a Modal Dependent Type Theory*

In a setting with Church-style abstraction, if one wishes to give the possibility for seemingly untyped abstraction, another mechanism has to be resorted to, typically elaboration using meta-variables. This is described in *e.g.* Asperti et al. [Asp+12], which combines a rule similar to Rule Abs – where the type of an abstraction is inferred – with another one, similar to the Curry-style one – where abstraction is checked, see opposite. While such a rule would make a system as that we have just described "over-complete", it is a useful addition to enable the propagation of checking information upwards in the derivation, which is crucial in elaboration phases, even in Church-style.

$$\frac{\begin{array}{c} T \to_{\mathrm{h}}^{\star} \Pi\, x\colon A'.B \\ \Gamma \vdash A \triangleright_{\square} \square_i \\ A \cong A' \qquad \Gamma, x\colon A \vdash t \triangleleft B \end{array}}{\Gamma \vdash \lambda\, x\colon A.\, t \triangleleft T}$$

4.2.3. Uniqueness

All the bidirectional judgements of Figure 4.2a are syntax-directed, in the sense that there is always at most one rule that applies to derive a certain typing judgement, given a fixed subject. But there is still some indeterminacy. Indeed, in rules involving reduction no strategy is fixed, thus two different reducts can be used with the same rule, resulting in different inferred types. However, inferred types are still related:

Theorem 4.4. Uniqueness of inferred type up to joinability

> If Γ is well-formed, $\Gamma \vdash t \triangleright T$ and $\Gamma \vdash t \triangleright T'$ then T and T' both reduce to a common T'', *e.g.* $T \to^{\star} T''$ and $T \to^{\star} T''$. In particular, $T \cong T'$.

Proof.

By mutual induction on the first derivation, together with the same property for constrained inference.

The main idea is to use confluence to relate different reduction paths in Rules ΠInf and UnivInf. For the other rules, the conclusion is direct from the induction hypotheses. □

Combining this with correctness and completeness, we get uniqueness of types for the undirected system.

Theorem 4.5. Uniqueness of types

> If $\Gamma \vdash t : T$ and $\Gamma \vdash t : S$ then $T \cong S$.

Proof.

Since $\Gamma \vdash t : T$, by correctness there exists some T' such that $\Gamma \vdash t \triangleright T'$ and moreover $T' \cong T$. Similarly, there exists some S' such that $\Gamma \vdash t \triangleright S'$ and moreover $S' \cong S$. But by uniqueness $T' \cong S'$, and thus $T \cong S$. □

In order to completely eliminate indeterminacy, a reduction strategy can be fixed. This amounts to replacing full reduction with weak-head reduction, *e.g.* to replace the two reduction rules in Figure 4.2b by those of Figure 4.3. This is still correct and complete. Correctness follows exactly the same proof as Theorem 4.1. As for completeness, the main point is to show an analogous to Lemma 4.2 for weak-head reduction.

$$\textsc{UnivWhInf}\ \frac{\Gamma \vdash t \triangleright T \qquad T \to_{\mathrm{h}}^{\star} \square_i}{\Gamma \vdash t \triangleright_{\square} \square_i}$$

$$\Pi\textsc{WhInf}\ \frac{\Gamma \vdash t \triangleright T \qquad T \to_{\mathrm{h}}^{\star} \Pi\, x{:}\, A.B}{\Gamma \vdash t \triangleright_{\Pi} \Pi\, x{:}\, A.B}$$

Figure 4.3. Constrained inference with a weak-head strategy

Theorem 4.6. Reduction strategy

> If Rules UnivInf and ΠInf are replaced by UnivWhInf and ΠWhInf, then given a well-formed context Γ and a term t there is at most one T such that $\Gamma \vdash t \triangleright T$, and at most one T' such that $\Gamma \vdash t \triangleright_h T'$.

Proof.

Once again, by mutual induction.

For inference, given a fixed term t there is always at most one rule which applies to derive $\Gamma \vdash t \triangleright T$, since there is exactly one rule per term former. Combining this with the uniqueness of types inferred in the premises by induction hypothesis is enough to conclude.

For the constrained inference judgement, once again there is only one rule that applies. Since weak-head reduction is deterministic – given T, there is at most one T' such that $T \to_{\mathrm{h}}^{1} T'$ –, there is at most one weak-head normal form $\square$ or $\Pi\, x{:}\, A.\ B$ for a type. Hence, the type obtained by constrained inference is unique. □

4.2.4. Strengthening

Reasoning on the bidirectional derivation makes proofs easier, while correctness and completeness ensure the results can be carried to the undirected system. One way to understand this is that the canonical derivation obtained by combining correctness and completeness is more structured, and thus more amenable to proofs.

An example of this is the strengthening property, a consequence of conditional stability under renaming. We explained in Section 3.4 why proving these in the undirected system is not straightforward: the issue is that computation is too unconstrained, so that derivations might make use of needless variables. Bidirectional typing, however, does not have this defect, since no type is ever "invented". Rather, they are obtained either by reduction of previously inferred types, or as inputs. This means that types in a bidirectional derivation never mention useless variables, and thus that the following holds.

Theorem 4.7. Conditional stability under renaming – bidirectional

> Whenever we have
>
> - $x_1 : A_1 \dots x_n : A_n \vdash t \triangleright T$
> - for all i such that x_i appears in t, there is a variable y_i such that $(y_i : A_i[x_1 \coloneqq y_1 \dots x_n \coloneqq y_n]) \in \Delta$
>
> it also holds that $\Delta \vdash t[x_1 \coloneqq y_1 \dots x_n \coloneqq y_n] \triangleright T[x_1 \coloneqq y_1 \dots x_n \coloneqq y_n]$.

Proof.

By a direct induction on the typing derivation.

Note that we do not even need Δ to be well-formed. □

And as a special case, strengthening follows.

Theorem 4.8. Strengthening – bidirectional

> Whenever $\Gamma, x : A \vdash t \triangleright T$ and x does not appear in t, $\Gamma \vdash t \triangleright T$ is derivable.

From those, conditional stability under renaming and strengthening for the undirected system can be obtained without any difficulty.

5. Bidirectional PCUIC

5.1 **Cumulativity** 61
5.2 **Inductive Types** 62
5.2.1 The pair type 62
5.2.2 Polymorphic inductive types . 63

As we have seen in Section 3.6, there is much more to the real Coq than CC_ω. The ideas exposed in the previous chapter nevertheless scale very well to these extensions. There are two areas, though, where some care needs to be taken. The first is cumulativity, which in particular forces us to reconsider the statement of the completeness and uniqueness properties, see Section 5.1. But the main one is the introduction of inductive types. In particular, there is a subtle interplay with cumulativity in the treatment of pattern-matching. Working on the formalized proof of completeness in MetaCoq led to the discovery of an incompleteness bug in the kernel of Coq linked to this. In Section 5.2 we show how the bidirectional setting adapts to inductive types, and try and give an intuition of the origin of the completeness issue.

We do not give precise proofs in this chapter, instead relying on the formalization in MetaCoq described in Part 'A Certified Kernel for Coq, in Coq'.

5.1. Cumulativity

The introduction of the more liberal cumulativity rules in the undirected system of course calls for an update to the computation rules. The change to Rule Check is direct: simply replace conversion with cumulativity, as done in Rule CheckCum opposite. As for the constrained inference rules, they do not even need any modification. Intuitively, this is because there is no reason to degrade a type to a larger one, unless it is forced by a given target type in the checking judgment.

$$\textsc{CheckCum}\ \frac{\Gamma \vdash t \triangleright T \qquad T \preceq T'}{\Gamma \vdash t \triangleleft T'}$$

The statement of completeness also needs to account for cumulativity, and becomes the following one.

Theorem 5.1. Completeness, with cumulativity

If $\Gamma \vdash t : T$, then $\Gamma \vdash t \triangleright T'$ is derivable for some T' such that $T' \preceq T$.

This also means that in the setting of PCUIC, uniqueness of types up to conversion is not true any more. For instance, we both have $\Gamma \vdash \square_0 : \square_1$ and $\Gamma \vdash \square_0 : \square_2$, but $\square_1$ and $\square_2$ are not convertible. In that context, however, the type $\square_1$ still has a special property: it is minimal among all types, what we call a principal type.

Definition 5.2. Principal type

The type T is a *principal type* for term t – in a context Γ – if $\Gamma \vdash t : T$ and for any T' such that $\Gamma \vdash t : T'$, we have $T \preceq T'$.

The existence of such a principal type is the same as uniqueness of types up to cumulativity. Moreover, even in the cumulative setting, Theorem 4.4[1] stays true. Intuitively, this is because it only relies on properties of reduction, but not of conversion. Thus, following the same proof as that of Theorem 4.5,[2] we obtain that inferred types are principal.

1: Uniqueness of inferred types up to joinability.

2: Uniqueness of types for undirected typing.

Theorem 5.3. Inferred types are principal

> If Γ is well-formed and $\Gamma \vdash t \triangleright T$, then T is a principal type for t in Γ.

Proof.

If $\Gamma \vdash t : T'$, then by completeness there exists some T'' such that $\Gamma \vdash t \triangleright T''$, and moreover $T'' \preceq T'$. But by Theorem 4.4, $T \cong T'' \preceq T'$ and thus $T \preceq T'$, and T is thus indeed a principal type for t in Γ. □

The existence of principal types is not so easy to prove directly, as it more or less amounts to showing correctness and completeness of the bidirectional system at once. Nevertheless, it is useful, because it in particular means that any well-typed term t has an unambiguous smallest universe, which can be obtained as the principal type of its principal type. This means that there is a good separation between irrelevant propositions – those terms whose smallest universe is Prop – and relevant terms – those whose smallest universe is some $\square_i$ –, and that this stays true even in presence of cumulativity, and even if $\mathrm{Prop} \preceq \square_i$. If this were not the case, the erasure of propositional content – which is one of the important use cases of Prop – would not make sense.

5.2. Inductive Types

5.2.1. An example: the pair type

To set ideas straight, let us look at how we can adapt the dependent pair type of Figure 3.12 to the bidirectional setting: see Figure 5.1. To obtain these rules, first notice that all undirected typing rules for the pair type (Figure 3.12) must become inference rules if we want the resulting system to be complete. The question therefore is once again to choose modes for the premises. Rules PairTy and PairInf are very similar to the rule for Π-types, there is not much surprise there.

Rule Pair shows why we insisted in the undirected system on recording the types A and B in the pair. Indeed, they are needed to know which type to

$$\textsc{PairTy}\ \frac{\Gamma \vdash A \triangleright_\square \square_i \qquad \Gamma, x : A \vdash B \triangleright_\square \square_j}{\Gamma \vdash \Sigma\, x : A.\, B \triangleright \square_{\max(i,j)}}$$

$$\textsc{Pair}\ \frac{\begin{array}{c}\Gamma \vdash A \triangleright_\square \square_i \\ \Gamma, x : A \vdash B \triangleright_\square \square_j \qquad \Gamma \vdash t \triangleleft A \qquad \Gamma \vdash u \triangleleft B[x := t]\end{array}}{\Gamma \vdash (t,u)_{(A,x.B)} \triangleright\ \Sigma\, x : A.\, B}$$

$$\textsc{PairInd}\ \frac{\begin{array}{c}\Gamma \vdash s \triangleright_\Sigma\ \Sigma\, x : A.\, B \qquad \Gamma, z : \Sigma\, x : A.\, B \vdash P \triangleright_\square \square \\ \Gamma, y_1 : A, y_2 : B[x := y_1] \vdash b \triangleleft P[z := (y_1, y_2)_{(A,x.B)}]\end{array}}{\Gamma \vdash \mathrm{ind}_\Sigma\,(s; z.P; y_1.y_2.b) \triangleright P[z := s]}$$

$$\textsc{PairInf}\ \frac{\Gamma \vdash t \triangleright T \qquad T \to^\star \Sigma\, x : A.\, B}{\Gamma \vdash t \triangleright_\Sigma\ \Sigma\, x : A.\, B}$$

Figure 5.1. Bidirectional pair type

infer for the pair. Without the annotation, one could infer a type A for t and a type B' for u, but there are potentially many incomparable types B that would be correct for the whole pair, depending on which instances of t in B' are abstracted to x. We only know that B' is $B[x := t]$, but this is not enough to inambiguously determine B. This impossibility to invert a substitution is a general source of need for annotations, which is not specific to pair types!

Finally, Rule PairInd is the most complex. In presentations of recursors, often the predicate appears first, then the branches, and finally the scrutinee. But this is not possible here, as the parameters of the inductive type are needed to construct the context in which the predicate is typed. Instead, those parameters can be inferred from the scrutinee. Thus, a type for the scrutinee is first obtained using a new constrained inference judgment, forcing the inferred type to be a Σ-type, but leaving its parameters free. Next, these parameters can be used to construct the context to type the predicate. And finally, once the predicate is known to be well-formed, it can be used to type-check the branch.

This same approach can be readily extended to the other inductive types of Section 3.5, with recursion or indices posing no specific problems.

5.2.2. Polymorphic inductive types

The account of general inductive types in PCUIC is slightly different from the one we just gave. The reason for this is that giving a general account of rules which infer type levels like our Rule PairTy is not easy. Indeed, the parameters of an inductive type can be of a type much more complex than simply $\Box$, and in that general setting deciding which type variable can be inferred is a non-trivial problem. Instead, the polymorphic inductive types as implemented in Coq store explicit universe levels on inductive types and constructors. The pair type of Figure 5.1, for instance, would contain universe levels i, j, so that both A and B would be checked rather than having their level inferred. The rule for the type constructor in that context is given opposite. This makes the treatment of complex inductive types possible by using checking uniformly – rather than relying on constrained inference to infer universe levels – at the cost of possibly needless annotations, as here with Σ-types. This is mostly invisible for the end user though, as she does very seldom write universe levels thanks to typical ambiguity anyway.

$$\frac{\Gamma \vdash A \triangleleft \Box_i \qquad \Gamma, x : A \vdash B \triangleleft \Box_j}{\Gamma \vdash \Sigma\,{}^{@i,j} x : A.\ B \triangleright \Box_{\max(i,j)}}$$

In the same spirit, pattern-matching in Coq – and its counterpart in PCUIC – also stores enough information to easily reconstruct the context in which the predicate and branches are typed. This information consists in universe levels – for polymorphic inductive types – and parameters of the inductive type. Thus, the actual typing rule for pattern-matching in the case of Σ-types is closer to the following one:

$$\frac{\begin{array}{c}\Gamma \vdash s \triangleright_{\Sigma}\ \Sigma\,{}^{@i,j} x : A.\ B \\ i \leq i' \qquad j \leq j' \qquad A \preceq A' \qquad B \preceq B' \qquad \Gamma, z : \Sigma\, x : A'.\ B' \vdash P \triangleright_{\Box} \Box \\ \Gamma, y_1 : A', y_2 : B'[x := y_1] \vdash b \triangleleft P[z := (y_1, y_2)_{(A', x.B')}]\end{array}}{\Gamma \vdash \mathrm{match}_{\Sigma\,;i',j';A',B'}(s; z.P; y_1.y_2.b) \triangleright P[z := s]}$$

Note that the domain and codomain are compared using cumulativity. This is crucial to retain subject reduction. Indeed, reduction of the scrutinee might make its inferred type decrease. For instance, suppose we have a polymorphic inductive $I^{@i}$ with a single constructor c such that $A : \square_i \vdash c^{@i}(A)$. Now consider

$$\left(\lambda\, y : I^{@1}.\ y\right)\ c^{@0}(\mathbf{N}) \to^1 c^{@0}(\mathbf{N})$$

the redex infers type $I^{@1}$, while the reduct infers $I^{@0}$. Thus, if such a term is plugged as scrutinee in a pattern-matching, the whole term is still typeable after the reduction of the scrutinee because we allow inequalities rather than equalities between levels.

But here lies a subtle issue: in pen-and-paper accounts of recursors, the predicate and branches are often represented respectively as Π-types and λ-abstractions. This is also how previous versions of Coq represented pattern-matching.[3] But recall that in PCUIC, cumulativity is equivariant on the domain of Π-types. This led to an implementation that wrongly compared the universe levels using equality rather than inequality, leading to a completeness bug that manifested as a failure of subject reduction in situations such as the one above.[4] This prompted subsequent work, both on the theory of PCUIC and on the implementation, to remove the use of Π- and λ-abstractions completely from pattern-matching[5], making both the implementation less ad-hoc, and the theory cleaner. A detailed summary has been given in Sozeau, Lennon-Bertrand, and Forster [SLF22].

Further investigations in that area might still be valuable though, in particular in order to determine what kind of annotations are actually needed for pattern-matching, both in theory and in practice. Can we give a presentation of polymorphic inductive types that is as lightweight as pair types in Figure 5.1? The bidirectional presentation is valuable there, because now it is clear what the specification of an alternative syntax is: it should remain complete, in the sense of Theorem 5.1.

3: Until version 8.14 to be precise.

4: A precise description of the problem in the kernel and an example similar to the one above are given in issue #13495.

5: This was carried out by Pierre-Marie Pédrot starting with pull-request #13563, following ideas that had been laid down earlier by Hugo Herbelin in the Coq enhancement proposal #34.

[SLF22]: Sozeau et al. (2022), *The Curious Case of Case: Correct & Efficient Representation of Case Analysis in Coq and MetaCoq*

6. Bidirectional Conversion

In Chapters 4 and 5, we considered typing, and saw how it could be turned into a bidirectional relation. However, we did not consider conversion. Indeed, since we chose to use an untyped notion of conversion, a bidirectional approach would not have made sense, as there was no type around in conversion.

However, the typed presentation of conversion is also a popular one, and in that setting the question of giving a bidirectional presentation *is* sensible. Luckily, such a presentation is already available if we go through the literature with the right glasses on. Indeed, in Abel, Öhman, and Vezzosi [AÖV17], decidability of conversion is shown by introducing a "conversion algorithm", a relation presented via inference rules, but which directly corresponds to an implementable convertibility check. This is somewhat similar to how we show decidability of typing in Part 'A Certified Kernel for Coq, in Coq' by going through bidirectional typing as an intermediate, more structured representation. But the interesting point is that this typed[1], algorithmic conversion is in fact bidirectional! Indeed, while regular conversion-checking uses the type as input, it is mutually defined with a specific relation to compare neutrals, which *infers* a type while checking that the neutrals are convertible. In this chapter, we re-cast the ideas of Abel, Öhman, and Vezzosi [AÖV17] in our setting, clearly delineating their bidirectional nature.

[AÖV17]: Abel et al. (2017), *Decidability of Conversion for Type Theory in Type Theory*

1: Type information is used to trigger η-expansion when comparing inhabitants of a Π-type.

Moreover, we can use that bidirectional structure to show that this typed algorithmic conversion agrees with an untyped one, close to the conversion algorithm implemented in Coq. This is interesting, because currently PCUIC as presented in MetaCoq is not able to handle extensionality rules such as the η-rule for functions. This is not because we do not know how to handle them in the kernel[2] but rather because it is difficult to give a good specification of them in the untyped setting chosen for MetaCoq's conversion.[3] Thus, showing that typed and untyped algorithms agree could be a first step towards a specification of MetaCoq using typed conversion, which would facilitate the incorporation of extensionality rules that are currently direly missing to the project.

2: Coq's kernel has an implementation that takes care of extensionality rules in a term-directed fashion.

3: The changelog of Coq 8.4, where extensionality for functions was introduced, actually reads: *"The addition of η-conversion is justified by the confidence that the formulation of the Calculus of Inductive Constructions based on typed equality (such as the one considered in Lee and Werner to build a set-theoretic model of CIC [LW11]) is applicable to the concrete implementation of Coq."* See Lennon-Bertrand [Len22] for more insight on the difficulties in the untyped setting.

[LW11]: Lee et al. (2011), *Proof-irrelevant model of CC with predicative induction and judgmental equality*

[Len22]: Lennon-Bertrand (2022), *À bas l'η – Coq's troublesome η-conversion*

The chapter is organized as follows: Section 6.1 introduces the main relation we will be interested in, namely the bidirectional conversion inspired by Abel, Öhman, and Vezzosi [AÖV17]; Section 6.2 presents its untyped counterpart, close to the implementation of Coq; Section 6.3 discusses the meta-theoretical properties needed for the rest of the chapter, and the difficulties they pose; finally, Section 6.4 presents the equivalence between this bidirectional conversion and the untyped one.

The content of this chapter is rather new, and its material has not yet been submitted to peer-reviewing. As such, it should be regarded as a first attempt at making interesting ideas visible, rather than a finished and polished exposition.

6.1. Bidirectional Conversion

6.1.1. Extensionality and η-rules

Before we can get to bidirectional conversion, let us first go over why using typed conversion is interesting. Typed conversion is as old as type theory itself [Mar72], and there are two main reasons that make it a better choice over untyped conversion as we have used until now. The first is that it is easier to build models[4] using typed conversion, because these can use that extra information to interpret conversion *at a given type*. But the reason that is of interest to us here, as we do not build such models, are extensionality rules.

4: Or logical relations, translations...

In general, extensionality rules allow equating two terms, not based on their shape,[5] but on their type. The most basic one is that for functions, which says that any function f and g of type $\Pi\, x\colon A.\, B$ should be convertible whenever $f\ x$ and $g\ x$ are – note that here f and g are *any* functions. As their name suggest, this kind of rules constrain the system to be somewhat extensional. For instance, in the case of functions, f and g cannot contain any "hidden" information other than their behaviour using application, because such information would disappear when applying the extensionality rule. In Coq, similar extensionality rules exist for dependent pair types[6], and more broadly for record types,[7] as well as for strict propositions [Gil+19; PT22].[8]

5: As is the case of all the rules introduced so far, especially β and ι.

6: Saying that p and q of type $\Sigma\, x\colon A.\, B$ are convertible whenever their two components are.

7: A generalized version of pair types, see Section 3.6.5.

8: Saying that whenever $P : \mathrm{SProp}$, and $p : P, q : P$, p and q are convertible.

In the case of functions,[9] the extensionality rule is inter-derivable with what is called the η-rule, which equates f and $\lambda\, x\colon A.f\ x$. While less useful than β-rules, η-rules are still valuable. For instance, in the setting of homotopy type theory, they are needed to deduce function extensionality from the univalence axiom [Uni13, Theorem 4.9.4]. Strict propositions are also seen as a promising tool for proof management [App22].

9: Something similar happens for record types.

[Uni13]: Univalent Foundation Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*

[App22]: Appel (2022), *Coq's Vibrant Ecosystem for Verification Engineering (Invited Talk)*

6.1.2. Conversion checks, neutral comparison infers

If we want to describe such type-based rules, it is natural to wish for a typing relation that maintains the type, in order to use it to trigger extensionality rules. This what happens for instance in Agda [Nor07].[10] A nice theoretical presentation of this is given by the "algorithmic conversion" of Abel, Öhman, and Vezzosi [AÖV17], from which we take inspiration here to describe a bidirectional conversion relation for CC_ω.

[Nor07]: Norell (2007), *Towards a practical programming language based on dependent type theory*

10: In the specific case of functions, for performance reasons the Agda implementation actually uses the same term-directed technique as Coq, similar to that of Section 6.2. But type-directed extensionality rules are used *e.g.* for the definitional unit type.

[AÖV17]: Abel et al. (2017), *Decidability of Conversion for Type Theory in Type Theory*

The important intuition about this relation is that it actually decomposes conversion in two components. On one side, *generic conversion*, that we will continue writing[11] $\cong$, which takes a type as input – *i.e.* it *checks*. On the other side, *neutral comparison*, written $\approx$, which takes a type as output – it *infers*. There are two reasons for this. First, applying extensionality rules on neutrals is useless, as this will simply create blocked redexes. For instance, if n and n' are neutral functions, $n\ x$ and $n'\ x$ are convertible exactly when n and n' are. But more importantly, the inferred information is used to know at which type the recursive appeals to conversion need to be done. In the case of applications again, comparing $n\ t$ with $n'\ t'$, we need to infer a type $\Pi\, x\colon A.\, B$ while recursively comparing n with n' to compare t to t' at type A. This information can only be inferred from the neutrals: even if we know

11: We use the colour blue for the typed relations, and the $\vdash_{\mathrm{t}}$ symbol to distinguish typing judgments defined using the typed relations.

that the comparison between $n\ t$ and $n'\ t'$ happens at type T, this gives no insight on the type at which t and t' should be compared.

$$\textsc{Check}\ \frac{\Gamma \vdash_{\mathrm{t}} t \triangleright T \qquad \Gamma \vdash T \preceq T' \triangleleft}{\Gamma \vdash_{\mathrm{t}} t \triangleleft T'} \qquad \textsc{RedCum}\ \frac{T \to_{\mathrm{h}}^{\star} T' \qquad U \to_{\mathrm{h}}^{\star} U' \qquad \Gamma \vdash T' \preceq_{\mathrm{h}} U' \triangleleft}{\Gamma \vdash T \preceq U \triangleleft}$$

Figure 6.1a. Generic cumulativity

We wish to extend CC_ω, so the rules we present here are meant to complement the rules of Figures 4.2a and 4.2b, replacing Rule Check of Figure 4.2b by Rule Check of Figure 6.1a. We cannot define a system based purely on conversion,[12] so we use *generic cumulativity* $\preceq$ instead. Note also that there is no known level at which the two types should be compared, hence generic cumulativity "checks", but against the mere fact of being a type, rather than against a precise type. This is akin to the relation written $\Gamma \vdash T \cong T'$ or $\Gamma \vdash T \cong T'\ \mathit{Type}$ often used in the setting of Martin-Löf type theory. To deduce generic cumulativity, there is only one rule that applies, Rule RedCum: both arguments are reduced to weak-head normal forms, before being compared by the auxiliary relation $\preceq_{\mathrm{h}}$.

12: This is due to the product rule, to which we will get soon.

$$\textsc{BdNeuCum}\ \frac{\Gamma \vdash N \approx N' \triangleright S}{\Gamma \vdash N \preceq_{\mathrm{h}} N' \triangleleft} \qquad \textsc{BdUniCum}\ \frac{i \leq j}{\Gamma \vdash \square_i \preceq_{\mathrm{h}} \square_j \triangleleft}$$

$$\textsc{BdΠCum}\ \frac{\Gamma \vdash A \cong A' \triangleleft \qquad \Gamma, x\colon A \vdash B \preceq B' \triangleleft}{\Gamma \vdash \Pi\, x\colon A.\ B \preceq_{\mathrm{h}} \Pi\, x\colon A'.\ B' \triangleleft}$$

Figure 6.1b. Generic cumulativity between reduced types

This auxiliary relation, in turn, is defined by the rules of Figure 6.1b, which either apply congruence rules if both types being compared are canonical forms (Rules BdUniCum and BdΠCum), or call neutral comparison otherwise (Rule BdNeuCum). In the latter case, we do not need to check that the type S inferred by the neutral comparision matches that at which cumulativity happens: this will always be true thanks to the well-typing invariants we maintain, so we do not need to re-check it here. Instead, the inferred type is only useful to recover information in further neutral comparison, see Figure 6.1d.

$$\textsc{RedConvTy}\ \frac{T \to_{\mathrm{h}}^{\star} T' \qquad U \to_{\mathrm{h}}^{\star} U' \qquad \Gamma \vdash T' \cong_{\mathrm{h}} U' \triangleleft}{\Gamma \vdash T \cong U \triangleleft} \qquad \textsc{BdNeuConvTy}\ \frac{\Gamma \vdash N \approx N' \triangleright S}{\Gamma \vdash N \cong_{\mathrm{h}} N' \triangleleft}$$

$$\textsc{BdUniConvTy}\ \frac{i = j}{\Gamma \vdash \square_i \cong_{\mathrm{h}} \square_j \triangleleft} \qquad \textsc{BdΠConvTy}\ \frac{\Gamma \vdash A \cong A' \triangleleft \qquad \Gamma, x\colon A \vdash B \cong B' \triangleleft}{\Gamma \vdash \Pi\, x\colon A.\ B \cong_{\mathrm{h}} \Pi\, x\colon A'.\ B' \triangleleft}$$

Figure 6.1c. Generic conversion between types

Generic conversion between types is defined in Figure 6.1c, in a way very similar to generic cumulativity.

Next, we get to neutral comparison, in Figure 6.1d. Neutrals are related exactly when they are the same variable, applied to two lists of recursively convertible arguments. The interesting rule is Rule AppComp, where we see

$$\textsc{VarComp}\ \frac{(x\colon A) \in \Gamma}{\Gamma \vdash x \approx x \triangleright A} \qquad \textsc{AppComp}\ \frac{\Gamma \vdash n \approx n' \triangleright_\Pi \Pi\, x\colon A.\, B \qquad \Gamma \vdash t \cong t' \triangleleft A}{\Gamma \vdash n\, t \approx n\, t' \triangleright B[x \coloneqq t]}$$

$$\textsc{RedComp}\ \frac{\Gamma \vdash n \approx n' \triangleright T \qquad T \to_{\mathrm{h}}^{\star} \Pi\, x\colon A.\, B}{\Gamma \vdash n \approx n' \triangleright_\Pi \Pi\, x\colon A.\, B}$$

Figure 6.1d. Neutral comparison

the behaviour described earlier: the domain of the inferred type for the neutral is used to compare the arguments.

$$\textsc{RedConvTm}\ \frac{t \to_{\mathrm{h}}^{\star} t' \qquad u \to_{\mathrm{h}}^{\star} u' \qquad A \to_{\mathrm{h}}^{\star} A' \qquad \Gamma \vdash t' \cong_{\mathrm{h}} u' \triangleleft A'}{\Gamma \vdash t \cong u \triangleleft A}$$

$$\textsc{BdNeuConvUni}\ \frac{\Gamma \vdash n \approx n' \triangleright S}{\Gamma \vdash n \cong_{\mathrm{h}} n' \triangleleft \square_i} \qquad \textsc{BdNeuConvNeu}\ \frac{\Gamma \vdash n \approx n' \triangleright S \qquad \mathrm{ne}\, N}{\Gamma \vdash n \cong_{\mathrm{h}} n' \triangleleft N}$$

$$\textsc{BdUniConvTm}\ \frac{i = j}{\Gamma \vdash \square_i \cong_{\mathrm{h}} \square_j \triangleleft \square_k} \qquad \textsc{BdΠConvTm}\ \frac{\Gamma \vdash A \cong A' \triangleleft \square_i \qquad \Gamma, x\colon A \vdash B \cong B' \triangleleft \square_i}{\Gamma \vdash \Pi\, x\colon A.\, B \cong_{\mathrm{h}} \Pi\, x\colon A'.\, B' \triangleleft \square_i}$$

Figure 6.1e. Generic conversion between terms

Finally, we are left with generic conversion between terms, which is called recursively by neutral comparison. The first set of rules, given in Figure 6.1e is very similar to the one for types. First, the two terms and the type at which they are compared are reduced, and the terms are then compared using the auxiliary relation $\cong_{\mathrm{h}}$ (Rule RedConvTm). If the terms are neutrals, neutral comparison is used (Rules BdNeuConvUni and BdNeuConvNeu). This is only possible if the type is a universe or a neutral. Indeed, to keep the relation deterministic, this rule cannot be applied at a Π-type, where extensionality *must* be used instead.

Otherwise, congruence rules must be used. In case the comparison happens at the universe, these are very similar to that for types (Rules BdUniConvTm and BdΠConvTm). Note, however, that in order to maintain the well-formation invariant mandated by McBride's discipline, we should only appeal to $\Gamma \vdash t \cong t' \triangleleft A$ when we know that both t and t' check against A. But in Rule BdΠConvTm, the domains and codomains might be at a universe level lower that i even if the whole product is at that level.[13] Thus, in order to recursively compare A to A' and B to B', we must know that they still check against $\square_i$, which requires cumulativity.

13: For instance, A might be $\square_0$ and B might be $\square_1$, so that $A \to B$ is at level 2 but A is at level 1.

$$\textsc{BdFunConv}\ \frac{\Gamma, x\colon A \vdash f\, x \cong f'\, x \triangleleft B}{\Gamma \vdash f \cong_{\mathrm{h}} f' \triangleleft \Pi\, x\colon A.\, B}$$

Figure 6.1f. Generic conversion between functions

The last rule is that for comparing two functions (Rule BdFunConv). In that case, an extensionality rule is directly applied without even looking at the two terms. There is thus no primitive congruence rule for λ-abstractions, but it is derivable,[14] because $(\lambda\, x\colon A.\, t)\, x \to_{\mathrm{h}}^{\star} t$, and so in case both f and

14: This is Lemma 6.6.

f' are abstractions, the recursive calls amount to comparing their bodies.

The rules as given directly translate to an algorithm, as they are nicely term- or type-directed, *i.e.* there is always at most one rule that applies to derive a judgment. Moreover, if in generic cumulativity and generic conversion we view all objects as inputs,[15] in neutral comparison the type is an output and all other objects are inputs, and in reduction $t \to_{\mathrm{h}}^{\star} t'$, t is an input and t' is an output, then all rules respect McBride's discipline.

15: The subject is the "computational content" of the judgment, *i.e.* whether the conversion/cumulativity holds. This is similar to the conversion judgments of general type theories [BHL20].

[BHL20]: Bauer et al. (2020), *A general definition of dependent type theories*

6.2. Untyped Presentation

In the presentation of Section 6.1, types are carried around, but almost never used. Indeed, only Rule BdFunConv really needs the type information to be applied. However, there is an alternative approach, used by the kernels of Coq and Agda, which avoids looking at types altogether by replacing the type-directed Rule BdFunConv with term-directed ones. As types are not maintained, there is also no point in maintaining the context either. Thus, this alternative conversion simply relates two terms: $\mathit{intro} * t \cong t'$.[16] Let us now spell out the rules for this alternative, untyped presentation.

16: We use the colour purple for untyped relations, and the $\vdash_{\mathrm{u}}$ symbol for typing judgments defined using those relations.

$$\textsc{CheckUty}\ \frac{\Gamma \vdash_{\mathrm{u}} t \triangleright T \qquad T \preceq T'}{\Gamma \vdash_{\mathrm{u}} t \triangleleft T'}$$

$$\textsc{RedCumUty}\ \frac{t \to_{\mathrm{h}}^{\star} t' \qquad u \to_{\mathrm{h}}^{\star} u' \qquad t' \preceq_{\mathrm{h}} u'}{t \preceq u}$$

$$\textsc{RedConvUty}\ \frac{t \to_{\mathrm{h}}^{\star} t' \qquad u \to_{\mathrm{h}}^{\star} u' \qquad t' \cong_{\mathrm{h}} u'}{t \cong u}$$

$$\textsc{BdNeuCumUty}\ \frac{n \approx n'}{n \preceq_{\mathrm{h}} n'} \qquad \textsc{BdNeuConvUty}\ \frac{n \approx n'}{n \cong_{\mathrm{h}} n'}$$

Figure 6.2a. Untyped cumulativity and conversion

The first rules of Figure 6.2a are similar to those for the typed variants: cumulativity can be used in checking, and terms are compared by first reducing them to weak-head normal form, and if they are neutrals the special neutral comparison is called.

$$\textsc{BdUniCumUty}\ \frac{i \leq j}{\square_i \preceq_{\mathrm{h}} \square_j} \qquad \textsc{BdΠCumUty}\ \frac{A \cong A' \qquad B \preceq B'}{\Pi\, x{:}\, A.\, B \preceq_{\mathrm{h}} \Pi\, x{:}\, A'.\, B'}$$

$$\textsc{BdUniConvUty}\ \frac{i = j}{\square_i \cong_{\mathrm{h}} \square_j} \qquad \textsc{BdΠConvUty}\ \frac{A \cong A' \qquad B \cong B'}{\Pi\, x{:}\, A.\, B \cong_{\mathrm{h}} \Pi\, x{:}\, A'.\, B'}$$

Figure 6.2b. Untyped bidirectional conversion for types

The rules for the comparison of types are given in Figure 6.2b, and are again close to those for the typed variant: there is a congruence rule for Π-types, and universes are convertible when their levels are in the right relation.

$$\textsc{VarCompUty}\ \frac{}{x \approx x} \qquad \textsc{AppCompUty}\ \frac{n \approx n' \qquad t \cong t'}{n\ t \approx n\ t'}$$

Figure 6.2c. Untyped neutral comparison

In the case of neutral comparison, the rules (Figure 6.2c) are even simpler than in the typed case, because there is no need for a special rule to reduce the type. Thus, there are only two rules, one for application and one base case for variables.

$$\textsc{BdAbsCong}\ \frac{t \cong t'}{\lambda\, x{:}\,A.\, t \cong_{\mathrm{h}} \lambda\, x{:}\,A'.\, t'}$$

$$\textsc{BdAbsNeu}\ \frac{t \cong n'\ x \qquad \mathrm{ne}\, n'}{\lambda\, x{:}\,A.\, t \cong_{\mathrm{h}} n'} \qquad \textsc{BdNeuAbs}\ \frac{n\ x \cong t' \qquad \mathrm{ne}\, n}{n \cong_{\mathrm{h}} \lambda\, x{:}\,A'.\, t'}$$

Figure 6.2d. Untyped, bidirectional conversion for functions

Finally, the interesting difference appears in Figure 6.2d. Here what was done using only one generic rule (Rule BdFunConv) is decomposed into four of them, depending on whether each function in weak-head normal form is a neutral or an abstraction. In case both are abstractions, the extensionality rules amounts to a congruence, *i.e.* Rule BdAbsCong.[17] In case both are neutrals, the extensionality rule only inserts a useless application to a variable, but neutral comparison can be directly used instead, by means of Rule BdNeuConvUty. The only situation where the extensionality rule is useful is when comparing a neutral to an abstraction. But in those cases, the information that the comparison happens at a function type and that the neutral needs to be η-expanded can be obtained from the abstraction. This is what the symmetric Rules BdAbsNeu and BdNeuAbs do.

17: If we maintain the invariant that both terms that are compared have a common type, then there is no need to compare the domains of the abstractions because they are always convertible.

6.3. McBride's Discipline

6.3.1. Modes for the relations

As we have seen in Section 4.1.1, for a bidirectional system to be well-behaved, it must preserve the well-formation of the objects it manipulates as an invariant, what we have called McBride's discipline. First, we need to distinguish subjects, inputs and outputs of the judgments. In all relations we just defined, the subject[18] is not a term as in typing, but rather whether a certain relation holds. As in the case of the typing relation, the context is always an input. In cumulativity, conversion and neutral comparison, the two terms are also inputs, since we wonder whether two *given* terms are related. This is contrast with reduction, where only the redex is an input, while the reduct is an output. This separation of modes between conversion/cumulativity and reduction already appeared in Section 4.1.1. Finally, as hinted by the use of the inference *versus* checking symbols, the type is an output in neutral comparison, while it is an input in conversion and cumulativity. As for the type-level relations[19] there is no real input, only the knowledge that the comparison happens at the type level, which is similar to performing the comparison at some $\square_i$ for an unspecified i.

18: That which is under scrutiny.

19: That is, $\Gamma \vdash T \preceq T' \triangleleft$ and consort.

With the modes set down, the following definitions of inputs and outputs well-formation are rather natural. The only maybe surprising point is that we express all those conditions in the typed variant. This way, we need only consider the meta-theory of one system – the one based on typed relations –, and can carry over all these properties to the other system after we have proven their equivalence.

Definition 6.1. Inputs well-formation – typed relations

We say that "inputs are well-formed" for one of the relations of Figures 6.1a to 6.1f to mean the following:

- in the case of $\Gamma \vdash t \cong t' \lhd T$ and of $\Gamma \vdash t \cong_{\mathrm{h}} t' \lhd T$, that $\vdash \Gamma$, that there exists i such that $\Gamma \vdash_{\mathrm{t}} T \rhd_{\square} \square_i$, and that $\Gamma \vdash_{\mathrm{t}} t \lhd T$ and $\Gamma \vdash_{\mathrm{t}} t' \lhd T$;
- in the case of $\Gamma \vdash T \cong T' \lhd$, $\Gamma \vdash T \cong_{\mathrm{h}} T' \lhd$, $\Gamma \vdash T \preceq T' \lhd$ and $\Gamma \vdash T \preceq_{\mathrm{h}} T' \lhd$, that $\vdash \Gamma$, and that there exist i and j such that $\Gamma \vdash_{\mathrm{t}} T \rhd_{\square} \square_i$ and $\Gamma \vdash_{\mathrm{t}} T' \rhd_{\square} \square_j$;
- in the case of $\Gamma \vdash n \approx n' \rhd S$ and of $\Gamma \vdash n \approx n' \rhd_{\Pi} S$, that $\vdash \Gamma$, and that there exists T and T' such that $\Gamma \vdash_{\mathrm{t}} n \rhd T$ and $\Gamma \vdash_{\mathrm{t}} n' \rhd T'$.[20]

20: Note that we do not *a priori* demand that S be related to T, as this is a well-formation property of the *output* S.

Definition 6.2. Inputs well-formation – untyped relations

We say that "inputs are well-formed" for one of the relations of Figures 6.2a to 6.2d to mean the following:

- in the case of $t \cong t'$ and of $t \cong_{\mathrm{h}} t'$, that there exists some Γ, T and i such that $\vdash \Gamma$, $\Gamma \vdash_{\mathrm{t}} T \rhd_{\square} \square_i$, $\Gamma \vdash_{\mathrm{t}} t \lhd T$ and $\Gamma \vdash_{\mathrm{t}} t' \lhd T$ hold;
- in the case of $n \approx n'$, that there exists some Γ, T and T' such that $\vdash \Gamma$, $\Gamma \vdash_{\mathrm{t}} n \rhd T$ and $\Gamma \vdash_{\mathrm{t}} n' \rhd T'$.

Moreover, we say that "inputs are well-formed types" in the case of $T \cong T'$, $T \cong_{\mathrm{h}} T'$, $T \preceq T'$ and $T \preceq_{\mathrm{h}} T'$, to mean the existence of Γ, i and j such that $\vdash \Gamma$, $\Gamma \vdash_{\mathrm{t}} T \rhd_{\square} \square_i$ and $\Gamma \vdash_{\mathrm{t}} T' \rhd_{\square} \square_j$.

Definition 6.3. Outputs well-formation

We say that "outputs are well-formed" for neutral comparison between two terms n and n' assumed to be well-typed to mean the following:

- in the case of $\Gamma \vdash n \approx n' \rhd T$, that $\Gamma \vdash_{\mathrm{t}} n \rhd T$ holds, and also $\Gamma \vdash_{\mathrm{t}} n' \rhd T'$, for some T' such that $\Gamma \vdash T \cong T' \lhd$;
- in the case of $\Gamma \vdash n \approx n' \rhd_{\Pi} \Pi\, x{:}\, A.\, B$, that $\Gamma \vdash_{\mathrm{t}} n \rhd_{\Pi} \Pi\, x{:}\, A.\, B$ holds, and moreover that $\Gamma \vdash_{\mathrm{t}} n' \rhd_{\Pi} \Pi\, x{:}\, A'.\, B'$ holds too, with some A and B such that $\Gamma \vdash \Pi\, x{:}\, A.\, B \cong \Pi\, x{:}\, A'.\, B' \lhd$.

6.3.2. Meta-theory of the bidirectional system

Let us now try and see what meta-theoretical properties we need of the typed system to show that its rules respect McBride's discipline.

In Rules RedCum, RedConvTy and RedConvTm, the well-formation of inputs to the last premise under the hypothesis that inputs to the conclusion

are well-formed is exactly subject reduction. In the case of a β-redex, subject reduction is equivalent to the following weak version of stability by substitution.

Property 6.4. Stability of typing by substitution

> If $\Gamma, x\colon A \vdash_{\mathrm{t}} t \triangleright T$ and $\Gamma \vdash_{\mathrm{t}} u \triangleleft A$ hold and their inputs are well-formed, then $\Gamma \vdash_{\mathrm{t}} t[x \coloneqq u] \triangleleft T[x \coloneqq u]$.

A similar property appears even more directly in the case of neutral comparison, this time regarding output well-formation in Rule AppComp. Indeed, in that case by output well-formation of in premises, we can assume that $\Gamma \vdash_{\mathrm{t}} n' \triangleright_{\Pi} \Pi\, x\colon A'.\, B'$, with $\Gamma \vdash \Pi\, x\colon A.\, B \cong \Pi\, x\colon A'.\, B' \triangleleft$, and we need to show that $\Gamma \vdash B[x \coloneqq t] \cong B'[x \coloneqq t'] \triangleleft$. Again, here we have a form of stability by substitution.

Property 6.5. Stability of conversion by substitution

> If $\Gamma, x\colon A \vdash t \cong t' \triangleleft T$ and $\Gamma \vdash u \cong u' \triangleleft A$ and their inputs are well-formed, then $\Gamma \vdash t[x \coloneqq u] \cong t'[x \coloneqq u'] \triangleleft T[x \coloneqq u]$.

However, here lies a difficulty: Property 6.5 implies normalization. To see why, a first remark: congruence of conversion holds for all canonical forms, respectively by Rules BdΠConvTm and BdUniConvTm, and by the following lemma.

Lemma 6.6. Congruence of abstraction

> If $\Gamma, x\colon A \vdash t \cong t' \triangleleft B$ then $\Gamma \vdash \lambda\, x\colon A.\, t \cong \lambda\, x\colon A'.\, t' \triangleleft \Pi\, x\colon A.\, B$.[21]

21: For the purpose of this congruence, there is no need for a relation between A and A', but for the inputs to the conclusion to be well-formed, we should also have $\Gamma \vdash A \cong A' \triangleleft$.

Proof.

First, conversion is stable by anti-reduction, *i.e.* if $\Gamma \vdash u_2 \cong u_2' \triangleleft U_2$ holds and $u_1 \to_{\mathrm{h}}^{\star} u_2$, $u_1' \to_{\mathrm{h}}^{\star} u_2'$ and $U_1 \to_{\mathrm{h}}^{\star} U_2$ then $\Gamma \vdash u_1 \cong u_1' \triangleleft U_1$. Indeed, if the former holds, it must be by an application of Rule RedConvTm, and so there are u_3 and u_3' and U_3 respective reducts of u_2, u_2' and U_2 such that $\Gamma \vdash u_3 \cong_{\mathrm{h}} u_3' \triangleleft U_3$. But then also $u_1 \to_{\mathrm{h}}^{\star} u_3$ and similarly for the other two, and so we can use again Rule RedConvTm.

Now, by an application of Rule BdFunConv, we only need to show that $\Gamma, x\colon A \vdash (\lambda\, x\colon A.\, t)\ x \cong (\lambda\, x\colon A'.\, t')\ x \triangleleft B$ holds, and we can use stability by anti-reduction to conclude. □

Moreover, if we assume Property 6.5, then congruence also holds for application.

Lemma 6.7. Congruence of application

> Assuming Property 6.5, if $\Gamma \vdash t \cong t' \triangleleft \Pi\, x\colon A.\, B$ and $\Gamma \vdash u \cong u' \triangleleft A$ and their inputs are well-formed, then also $\Gamma \vdash t\ u \cong t'\ u' \triangleleft B[x \coloneqq u]$.

Proof.

The only way to obtain the first premise is to apply Rule RedCum and Rule BdFunConv. Thus, we have that $t \to_{\mathrm{h}}^{\star} f, t' \to_{\mathrm{h}}^{\star} f'$ and $\Gamma, x : A \vdash f\ x \cong f'\ x \lhd B$. By Property 6.5, we have

$$\Gamma \vdash (f\ x)[x \coloneqq u] \cong (f'\ x)[x \coloneqq u'] \lhd B[x \coloneqq u]$$

But since we assume no shadowing happens, x does not appear in f or f',[22] so that we actually have $\Gamma \vdash f\ u \cong f'\ u' \lhd B[x \coloneqq u]$. Now stability by anti-reduction is enough to conclude, since $t\ u \to_{\mathrm{h}}^{\star} f\ u$ and $t'\ u' \to_{\mathrm{h}}^{\star} f'\ u'$. □

22: In de Bruijn indices, f and f' are lifted when they are η-expanded, thus they cannot mention variable 0 corresponding to x.

Applying all these congruences in the diagonal case, we obtain reflexivity of conversion.

Proposition 6.8. Reflexivity

> Assuming Property 6.5, if $\vdash \Gamma$ and $\Gamma \vdash_{\mathrm{t}} t \rhd T$, then also $\Gamma \vdash t \cong t \lhd T$.

Proof.

By induction on the typing derivation, using the previous congruences in each case. □

But since conversion amounts to iterated weak-head normalization of both terms, reflexivity implies normalization, in the following sense.

Proposition 6.9. Normalization

> Assuming Property 6.5, if $\Gamma \vdash_{\mathrm{t}} t \rhd T$ and $\vdash \Gamma$, then there is some normal form t' such that $t \to^{\star} t'$.

Thus, if we wished to establish that our rules respect McBride's discipline, we would need a proof technique able to show normalization of the system under consideration. In the case of a system such as that of this chapter, a technique close to the logical relation of [AÖV17] might be enough. But if we add an impredicative sort of propositions, proofs of normalization are scarcer and further from the presentations of this chapter [Wer94; Alt93]. An alternative solution, following MetaCoq, would be to assume a property such as normalization and derive the needed meta-theory from that single assumption [Soz+20].

[AÖV17]: Abel et al. (2017), *Decidability of Conversion for Type Theory in Type Theory*

[Wer94]: Werner (1994), *Une Théorie des Constructions Inductives*

[Alt93]: Altenkirch (1993), *Constructions, Inductive Types and Strong Normalization*

[Soz+20]: Sozeau et al. (2020), *Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq*

In any case, a substantial meta-theoretical study would be needed, one that I do not wish to pursue further here. Thus, let us simply *assume* the properties we need for McBride's discipline to be correctly maintained in both presentations. Apart from stability by substitution and subject reduction that we have already mentioned, the main needed properties are those necessary to handle the left bias of rules. For instance, in Rule BdΠCum, the context is extended with some A, but we only know that B' is a type in a context extended by A', which is convertible to A. Similarly, in the second premise of Rule AppComp, the recursive conversion happens at type A, but t' is only known to check against some A' which is convertible to A.

Property 6.10. Subject reduction

> If $\Gamma \vdash_{\mathrm{t}} t \lhd T$ and $t \to_{\mathrm{h}}^{\star} t'$ then $\Gamma \vdash_{\mathrm{t}} t' \lhd T$.

Property 6.11. Stability by context and type conversion

> Let Γ and Γ' be two well-formed context that are pointwise convertible and T, T' be two well-formed types – respectively in Γ and Γ' –, such that $\Gamma \vdash T \cong T' \lhd$. If $\Gamma \vdash_{\mathrm{t}} t \lhd T$ then $\Gamma' \vdash_{\mathrm{t}} t \lhd T'$, and if $\Gamma \vdash_{\mathrm{t}} U \rhd \square_i$ then $\Gamma' \vdash_{\mathrm{t}} U \rhd \square_i$.

Conjecture 6.12. Meta-theoretical properties

> Properties 6.4, 6.5, 6.10 and 6.11 hold.

With this conjecture in hand, we can show that McBride's discipline is preserved, giving the following.

Proposition 6.13. Input well-formation – untyped

> If one of the relations of Figures 6.1a to 6.1f holds and its inputs are well-formed, then inputs to any sub-derivation are also well-formed and outputs are too.

Proof.

The proof is by mutual induction. It requires stability of typing by context/type cumulativity to handle the fact that the rules are left biased — *e.g.* context extension in Rule BdΠConvTy is done using the domain of the left Π-type –, and to deduce that the η-expansions of Rule BdFunConv are well-formed. Subject reduction is needed to know that weak-head reduction preserves well-formation. Finally, well-formation of outputs is necessary in Rule AppComp to ensure that t' indeed checks against A. It requires stability by substitution of conversion to ensure that outputs of Rule AppComp are well-formed. □

Proposition 6.14. Input well-formation – untyped

> If one of the relations of Figures 6.2a to 6.2d holds and its inputs are well-formed, then inputs to any sub-derivation are also well-formed and outputs are too.

Proof.

The proof is by mutual induction, and similar to the typed case. □

6.4. Equivalence of the presentations

With the meta-theoretical requirement exposed, we can now turn to the part of interest to us: the equivalence between both presentations.

Typed to untyped Unsurprisingly, the main rule that needs looking at is that which differs between the two systems, *i.e.* Rule BdFunConv. This is taken care of by the following lemma.

Lemma 6.15. Injectivity of η-expansion

> If $\vdash \Gamma, \Gamma \vdash_{\mathrm{t}} f \lhd \Pi x{:}A.\, B$ and $\Gamma \vdash_{\mathrm{t}} f' \lhd \Pi x{:}A.\, B$ hold, and moreover $f\ x \cong f'\ x$ holds too, then $f \cong_{\mathrm{h}} f'$.

Proof.

By inversion on the last hypothesis, we know that $f\ x$ and $f'\ x$ reduce to weak-head normal forms, say $f\ x \to_{\mathrm{h}}^{\star} v$, $f'\ x \to_{\mathrm{h}}^{\star} v'$ and that $v \cong_{\mathrm{h}} v'$. By inversion on the reductions, we get that also f and f' reduce to weak-head normal forms, say $f \to_{\mathrm{h}}^{\star} w$ and $f' \to_{\mathrm{h}}^{\star} w'$. Moreover, because of input well-formation and subject reduction, we know that both w and w' check against $\Pi\, x{:}\, A.\ B$. Since they are weak-head normal forms, they must thus be either λ-abstractions, or neutrals. We thus have four cases to consider.

In case both w and w' are λ-abstractions, say respectively $\lambda\, x{:}\, A.\ t$ and $\lambda\, x{:}\, A'.\ t'$, we have that $f\ x \to_{\mathrm{h}}^{\star} w\ x \to_{\mathrm{h}}^{1} t$, and similarly $f'\ x \to_{\mathrm{h}}^{\star} t'$. Because weak-head reduction is deterministic, we must have $t \to_{\mathrm{h}}^{\star} v$ and $t' \to_{\mathrm{h}}^{\star} v'$, but then since $v \cong_{\mathrm{h}} v'$ we also have $t \cong t'$. Thus, we can apply Rule BdAbsCong and conclude.

In case w is a λ-abstraction, say $\lambda\, x{:}\, A.\ t$ and w' is a neutral n', then v' must be equal to $n'\ x$. Then we have $f\ x \to_{\mathrm{h}}^{\star} w\ x \to_{\mathrm{h}}^{1} t \to_{\mathrm{h}}^{\star} v$, and thus $t \cong n'\ x$ since $v \cong_{\mathrm{h}} n'\ x$. Therefore, Rule BdAbsNeu applies to conclude. The reasoning in the symmetric case where w' is an abstraction and w is neutral is similar.

In the last case, both w and w' are neutrals, say n and n'. Then v and v' are respectively $n\ x$ and $n'\ x$. Since $n\ x \cong_{\mathrm{h}} n'\ x$, we must have also $n\ x \approx n'\ x$ because all rules but Rule BdNeuConvUty equate canonical forms. But then the last rule that applies must have been Rule AppComp, and thus we have $n \approx n'$. From this, we can get $n \cong_{\mathrm{h}} n'$ and since $f \to_{\mathrm{h}}^{\star} n$ and $f' \to_{\mathrm{h}}^{\star} n'$, we finally obtain $f \cong f'$, as expected. □

Theorem 6.16. Typed to untyped bidirectional conversion

The following implications hold whenever inputs are well-formed:

- if $\Gamma \vdash t \cong t' \triangleleft T$ or $\Gamma \vdash t \cong t' \triangleleft$, then $t \cong t'$;
- if $\Gamma \vdash T \preceq T' \triangleleft$, then $T \preceq T'$;
- if $\Gamma \vdash t \cong_{\mathrm{h}} t' \triangleleft T$ or $\Gamma \vdash t \cong_{\mathrm{h}} t' \triangleleft$ then $t \cong_{\mathrm{h}} t'$;
- if $\Gamma \vdash n \approx n' \triangleright T$ or $\Gamma \vdash n \approx n' \triangleright_{\Pi} T$ then $n \approx n'$.

Proof.

Once again, by mutual induction.

Most cases are direct, the induction hypothesis can directly be combined to give the desired result, replacing a typed rule by its untyped counterpart. The only difficulty is for the one rule which does not have an untyped counterpart, namely Rule BdFunConv. But in that case, Conjecture 6.12 ensures that inputs are well-formed since we started from a rule with well-formed inputs, thus Lemma 6.15 applies, giving the desired result. □

From untyped to typed Here again, the main point is to show that the rules of Figure 6.2d can be simulated by Rule BdFunConv. Lemma 6.6 already gives the congruence of abstractions, corresponding to Rule BdAbsCong. In the case of Rule BdAbsNeu – and its symmetric Rule BdNeuAbs–, it is also rather direct.

Lemma 6.17. Neutral against abstraction

If $\Gamma \vdash t \cong n'\ x \triangleleft B$, $\vdash \Gamma$, and there exists a T such that $\Gamma \vdash_{\mathrm{t}} \lambda\, x{:}A.\ t \triangleleft T$, then $T \to^\star \Pi\, x{:}A'.B'$ and $\Gamma \vdash \lambda\, x{:}A.\ t \cong_{\mathrm{h}} n' \triangleleft \Pi\, x{:}A'.\ B'$.

Proof.

By inversion on $\Gamma \vdash_{\mathrm{t}} \lambda\, x{:}A.\ t \triangleleft T$, we get that T must be convertible to the type inferred for $\lambda\, x{:}A.\ t$. But that inferred type is a Π-type, so T must also reduce to a Π-type. An application of Rule BdFunConv and stability of conversion by anti-reduction is enough to get $\Gamma \vdash \lambda\, x{:}A.\ t \cong_{\mathrm{h}} n' \triangleleft \Pi\, x{:}A'.\ B'$ from the first hypothesis. □

[AÖV17]: Abel et al. (2017), *Decidability of Conversion for Type Theory in Type Theory*

But the main difficulty comes from Rule BdNeuConvUty. Indeed, this rule can be applied whenever the compared terms are neutral while in the typed relation, following Abel, Öhman, and Vezzosi [AÖV17], extensionality for functions takes precedence over neutral comparison at Π-types. Thus, to simulate Rule BdNeuConvUty we need to show neutral comparison is always included in conversion, even if neutrals get η-expanded.

Lemma 6.18. Conversion subsumes neutral comparison

If $\Gamma \vdash n \approx n' \triangleright S$ and $\Gamma \vdash S \preceq T \triangleleft$ hold with well-formed inputs, then $\Gamma \vdash n \cong n' \triangleleft T$.

Proof.

By induction on the cumulativity hypothesis.

Both S and T reduce respectively to S' and T', and then one of the three rules of Figure 6.1b applies: S' and T' are either both neutrals, both universes, or both product types. In the first two cases, we are in a base case: either Rule BdNeuConvUni or Rule BdNeuConvNeu applies. In the last case, however, only Rule BdFunConv applies, *i.e.* the neutrals get η-expanded. Thus, S' is some $\Pi\, x{:}A.\ B$, and T' is some $\Pi\, x{:}A'.\ B'$. But then we still have $\Gamma, x{:}A \vdash n\ x \approx n'\ x \triangleright B$, so the induction hypothesis on the codomains can be used to conclude. □

We now have all ingredients for the second implication.

Theorem 6.19. Untyped to typed bidirectional conversion

If inputs are well-formed, then the following implications hold, with Γ and T being the respective context and type of the input well-formation hypothesis:

- if $t \cong t'$ then $\Gamma \vdash t \cong t' \triangleleft T$;
- if $t \cong_{\mathrm{h}} t'$ and T is a weak-head normal form, then $\Gamma \vdash t \cong_{\mathrm{h}} t' \triangleleft T$;
- if $n \approx n'$ then $\Gamma \vdash n \approx n' \triangleright T$.

If inputs are well-formed types, then the following implications hold, with Γ the context of the input well-formation hypothesis:

- if $T \cong T'$ then $\Gamma \vdash T \cong T' \triangleleft$;
- if $T \cong_{\mathrm{h}} T'$ then $\Gamma \vdash T \cong_{\mathrm{h}} T' \triangleleft$;
- if $T \preceq T'$ then $\Gamma \vdash T \preceq T' \triangleleft$;
- if $T \preceq_{\mathrm{h}} T'$ then $\Gamma \vdash T \preceq_{\mathrm{h}} T' \triangleleft$.

Proof.

By mutual induction. Most rules can be directly replaced by (one of) their typed counterpart, but for those which do not have such a counterpart, namely those of Figure 6.2d, and Rule BdNeuConvUty in case its arguments are terms – if they are types, then Rule BdNeuCum always applies. In each case, one of Lemmas 6.6, 6.17 and 6.18 is enough to conclude. □

A Certified Kernel for Coq, in Coq

Coq is a very complex tool. Even its kernel, which is only but a very small fraction of it, is already quite complex: it relies on subtle implicit invariants, which might not be properly maintained, especially when the code evolves. In practice, around one critical bug is found every year.[1] Although it is in practice generally difficult to exploit these and actually derive an inconsistency, even less so inadvertently, simply relying on the De Bruijn criterion[2] is not enough if one wants to trust Coq. Indeed, while CIC is well-understood and has been widely studied, this is much less true of the type theory actually implemented, PCUIC. Bugs therefore often creep in with the extra level of complexity coming with the implementation, rather than being the consequence of a defect of pen-and-paper proofs.[3]

1: A compilation of those is maintained by Coq's development team.

2: Keeping a small, trusted kernel that is the only one responsible for the validity of proofs.

3: This is for instance the case of the completeness issue exposed in Section 5.2.

These difficulties beg for a precise investigation of PCUIC, from the heights of the type system's meta-theory, all the way down to the sophisticated details of the implementation. Due to the complexity of the endeavour, it is not feasible on paper. Nor is it desirable: if in the end we wish to implement a certified kernel, it is natural to do so in a proof assistant, so that we can run that certified implementation. The natural framework is thus the MetaCoq project, which aims at giving tools to reify and manipulate Coq terms[4] inside Coq itself. This gives the possibility to write down and certify all kinds of procedures operating on these terms, the first to come to mind being of course a type-checker. This way, we can have both the help and guarantees offered by formal proofs inside a proof assistant, and the possibility to execute our implemented kernel.

4: Or, maybe more accurately, Gallina.

There are two important caveats to this, though. The first pertains to Gödel's second incompleteness theorem. Because of it, it is impossible to prove Coq's consistency inside Coq itself, meaning that the meta-theoretical study can only be partial, since otherwise it would allow a proof of consistency contradicting Gödel's theorem. In MetaCoq, this blind spot manifests as an axiom assuming the normalization of PCUIC, on which parts of the development relies. The second caveat is that writing down a certified kernel is not enough. Indeed, executing directly such a kernel in Coq would be much too slow to actually type-check any reasonably-sized term. Rather, we must rely on extraction, a procedure which erases the proof-related content of a certified program to only keep the algorithmically relevant one. As this erasure itself is a complex transformation, MetaCoq also incorporates a certified erasure procedure.

In this part of the thesis, I shall describe the portion of MetaCoq which is relevant to it. Chapter 7 gives a general overview of the meta-theory of PCUIC, with the main definitions, properties, and proof ideas. My technical contributions to this part of the development is relatively minor, mainly consisting of small patches. However, since I rely on that formalization in my main contributions, it seems fitting to go over it.

Chapter 8 concentrates on the formalization of bidirectional typing, as presented in Part 'Bidirectional Calculus of Inductive Constructions', and on the proof of correctness and completeness of the kernel implementation based on it. This is my main technical contributions to the MetaCoq project.

Although I will not describe it here, there is more to MetaCoq. The two main components I will omit are Template Coq, and the certified extraction procedure. The first faithfully represents the actual abstract syntax tree of Gallina and a typing predicate for it, gives a translation to the syntax used in the main theoretical development of PCUIC,[5] and shows that

5: Described in Figure 7.1.

both notions of typing are equivalent. It also provides facilities for quoting and unquoting of terms from Coq to MetaCoq's AST and back, in order to provide the possibility to write operation on Coq terms directly in MetaCoq – including, of course, the certified kernel of Chapter 8. The second component aims at certifyng the extraction procedure, relating the semantics of the original and extracted programs. The goal is to be able to extract the certified type-checker itself to an efficient one – execution in Coq is too inefficient if we wish to type-check realistic examples –, but also more generally to improve Coq's current extraction.

Throughout the part, source files of the MetaCoq project and specific definitions or theorems are referenced respectively as follows: PCUICTyping, and `typing`. They link directly to the source code of the project on GitHub – on a branch dedicated to this thesis.

7. Formalized Meta-Theory of PCUIC

7.1 Setting up the Definitions . . 83
7.1.1 Terms 83
7.1.2 Cumulativity 85
7.1.3 Typing 87
7.2 Stabilities 90
7.3 Confluence 91
7.3.1 Parallel reduction 92
7.3.2 Formalizing Takahashi's proof 93
7.4 The Road to Subject Reduction 93
7.4.1 Algorithmic cumulativity . . . 93
7.4.2 Reaping the fruits 94
7.5 Normalization 95
7.5.1 An abstract guard condition . 95
7.5.2 The normalization axiom . . . 95

Before we can attempt to build a certified kernel, we need a thorough meta-theoretical study of the type system. This is necessary in order to show that the invariants used by the kernel – typically, well-formation of the objects it manipulates – are preserved during the type-checking algorithm. The use of these invariants goes beyond correctness: the cumulativity test used as a sub-routine by the kernel needs to reduce terms, and, since all functions in Coq must be terminating, this reduction is defined by well-founded induction on the normalization of well-typed terms. Since evaluation is not normalizing on ill-typed terms, the mere *definition* of the cumulativity check relies on subject reduction to be able to iterate reduction steps.

The properties under scrutiny in this chapter are not new, and neither are the basic strategy of most proofs. Indeed, the development roughly follows the architecture we already exposed in Section 3.4. The main difficulty is the scale: due to the complexity of PCUIC, even well-understood techniques are challenging to apply. Moreover, subtleties that do not appear in a simpler setting become apparent – typically pertaining to universe levels or general inductive types –, demanding original ideas. Thus, rather than getting lost in the gory details of the formalization which are best understood by looking at it – and maybe replaying it –, we try and focus on describing these interesting subtleties.

In more details, we start with the main definitions : the syntax, cumulativity and typing judgments (Section 7.1). We follow with the basic stability properties (Section 7.2): renaming, substitution, environment extension, etc. Next comes the first important proof, that of confluence, and its multiple consequences (Section 7.3). This leads to the properties pertaining to typing, culminating with subject reduction (Section 7.4). Finally, we discuss the place of normalization (Section 7.5).

7.1. Setting up the Definitions: Terms, Cumulativity and Types

7.1.1. Terms

First thing first: the syntax of terms, defined in PCUICAst and reproduced in Figure 7.1.

It of course contains the term formers introduced in Chapter 3: `tRel` for variables, `tAbs` for abstractions, `tApp` for application, and `tProd` for dependent function types. The syntax uses De Bruijn indices for binders – the integer argument of the `tRel` term former –, but names are still recorded, mainly for printing purposes, directly in the binders – the `aname` argument of `tProp` and `tAbs`. There are also local definitions, in the form of the `tLetIn` constructor, binding the term `b` of type `B` in `t`.

The `tSort` constructor is for sorts – what we have called universes earlier. Its `Universe.t` argument represents its universe level, which can be either

```
Inductive term :=
| tRel (n : ℕ) (* Variable *)
| tVar (i : ident) (* Free named variables (e.g. in a goal) *)
| tEvar (n : ℕ) (l : list term) (* Existential variables *)
| tSort (u : Universe.t) (* Universe *)
| tProd (na : aname) (A B : term) (* Dependent function type *)
| tLambda (na : aname) (A t : term) (* Abstraction *)
| tLetIn (na : aname) (b B t : term) (* Local definition *)
| tApp (u v : term) (* Application *)
| tConst (k : kername) (ui : Instance.t) (* Constant *)
| tInd (ind : inductive) (ui : Instance.t) (* Inductive type *)
| tConstruct (ind : inductive) (n : ℕ) (ui : Instance.t) (* Constructor *)
| tCase (indn : case_info) (p : predicate term)
    (c : term) (brs : list (branch term)) (* Pattern-matching *)
| tProj (p : projection) (c : term) (* Primitive projection *)
| tFix (mfix : mfixpoint term) (idx : ℕ) (* Fixpoint *)
| tCoFix (mfix : mfixpoint term) (idx : ℕ). (* Co-Fixpoint *)
```

Figure 7.1. The Abstract Syntax Tree of terms in MetaCoq (`term`)

Prop or an algebraic expression based on universe variables, in order to handle typical ambiguity.

Next come `tVar` and `tEvar`, which correspond respectively to named variables and existential variables. These are ill-typed in the current notion of typing, and thus ignored in most of the development. Still, they are kept to be as faithful as possible to the representation of the Coq kernel. Indeed, the inductive `term` corresponds directly to the `constr` datatype used there.[1]

1: The only differences are that the latter uses an n-ary application rather than a binary one, and casts that inform the kernel as to which cumulativity algorithm to use, but which is left out since we implement only one such algorithm.

Follow the three term formers `tConst`, `tInd` and `tConstruct` all referring to previous definitions, stored in a global environment. The first corresponds to constants, that either have a body – definitions – or do not – axioms. The next two are respectively for inductive types and inductive constructors. Co-inductive and record types and constructors are also represented by these term formers, the information contained in the `inductive` argument is used to separate between them. All of these can be universe polymorphic, in which case they must be instantiated with a list of universes – their `Instance.t` argument.

The two subsequent `tCase` and `tProj` are destructors for (co-)inductive types. The latter is a projection, used to destruct record types. The former represents the pattern-matching construction. Its main components are the predicate `p`, the scrutinee `c` and the branches `brs`. While it will appear more clearly when giving the typing rule, let us note already that `p` and `brs` both contain not only the body of the predicate/scrutinee, but also the context extension over which they live, roughly corresponding to the variable bounds in the recursors of Section 3.5. Thus, they represent a form of binding in the "primitive" way of a context extension, rather than using Π-types or λ-abstraction. This is the new case representation alluded to at the end of Section 5.2.

Finally, the two very similar `tFix` and `tCoFix` are for (co-)fixed-points. These can be mutual: the `mfix` argument is a list of definitions, that can refer to each other.

7.1.2. Cumulativity

The next important definition is that of cumulativity, given in PCUICCumulativitySpec.[2] It is stated in the declarative untyped fashion, akin to how we defined declarative conversion in Figures 3.5b to 3.5d. This time, however, it is done relatively to both a global environment Σ and a context Γ, as these contain definitions that cumulativity can unfold.

2: The "Spec" part comes from the fact that this is the *specification* of cumulativity, by contrast to the algorithmic version encountered later on.

Cumulativity is defined mutually with conversion, because for instance when two Π-types are compared for cumulativity, their codomains are recursively compared for cumulativity, but their domains are compared for conversion instead.[3] Since the two relations are extremely similar, they are actually fused in a single inductive relation, `Σ ;;; Γ ⊢ t ≤s[pb] u`. This relation is indexed by a *conversion problem* `pb : conv_pb`, which can take the two values `Conv` and `Cumul`, so that cumulativity is actually `≤s[Cumul]` – and conversion is `≤s[Conv]`. This has the advantage that a lot of definitions and proofs can be factored using `conv_pb`. Moreover, using a simple boolean allows for case reasoning when needed, which would be more complex if the index was *e.g.* a relation.[4]

3: As explained in Section 3.6, this is a consequence of the models justifying cumulativity.

4: This used to be the case prior to the uniform introduction of `conv_pb`: the relation was the one to be used at leaves to compare universes, which differed between conversion and cumulativity.

```
| cumul_Trans : forall t u v,
  is_closed_context Γ → is_open_term Γ u →
  Σ ;;; Γ ⊢ t ≤s[pb] u →
  Σ ;;; Γ ⊢ u ≤s[pb] v →
  Σ ;;; Γ ⊢ t ≤s[pb] v
| cumul_Sym : forall t u,
  Σ ;;; Γ ⊢ t ≤s[Conv] u →
  Σ ;;; Γ ⊢ u ≤s[pb] t
| cumul_Refl : forall t,
  Σ ;;; Γ ⊢ t ≤s[pb] t
```

Figure 7.2a. Pre-order rules (`cumulSpec0`)

The first set of rules are the pre-order rules of Figure 7.2a: transitivity, symmetry and reflexivity. Note that symmetry restricts the conversion problem, since only conversion should be symmetric. Using this rule twice shows that conversion is included inside cumulativity. Another important thing to note is that transitivity is somewhat restricted: the middle term is required to be *well-scoped*,[5] *i.e.* all its variables refer correctly to either a binder or to the context Γ. This is key when proving the equivalence between this notion of cumulativity and the algorithmic version that appears later on. Indeed, this equivalence relies on confluence, which is only true on well-scoped terms in PCUIC. Thus, we need to know that declarative cumulativity only ever goes through well-scoped terms, which is exactly what this condition enforces.

5: Expressed by the `is_open_term` predicate.

```
| cumul_Prod : forall na na' a a' b b',
  eq_binder_annot na na' →
  Σ ;;; Γ ⊢ a ≤s[Conv] a' →
  Σ ;;; Γ ,, vass na a ⊢ b ≤s[pb] b' →
  Σ ;;; Γ ⊢ tProd na a b ≤s[pb] tProd na' a' b'
```

Figure 7.2b. Example of congruence rule (`cumulSpec0`)

Next come the rules of congruence. There are actually two kinds of them. The first are the "standard" ones, similar to those of Figure 3.5d. An example is given in Figure 7.2b. More interesting are the rules of Figure 7.2c, which implement "real" cumulativity. Rule `cumul_Sort` directly implements subtyping between universes, while rules `cumul_Ind` and `cumul_Construct`

```
| cumul_Ind : forall i u u' args args',
  cumul_Ind_univ Σ pb i #|args| u u' →
  All2 (fun t u ⇒ Σ ;;; Γ ⊢ t ≤s[Conv] u) args args' →
  Σ ;;; Γ ⊢ mkApps (tInd i u) args ≤s[pb] mkApps (tInd i u') args'
| cumul_Construct : forall i k u u' args args',
  cumul_Construct_univ Σ pb i k #|args| u u' →
  All2 (fun t u ⇒ Σ ;;; Γ ⊢ t ≤s[Conv] u) args args' →
  Σ ;;; Γ ⊢ mkApps (tConstruct i k u) args
     ≤s[pb] mkApps (tConstruct i k u') args'
| cumul_Sort : forall s s',
  compare_universe pb Σ s s' →
  Σ ;;; Γ ⊢ tSort s ≤s[pb] tSort s'
| cumul_Const : forall c u u',
  R_universe_instance (compare_universe Conv Σ) u u' →
  Σ ;;; Γ ⊢ tConst c u ≤s[pb] tConst c u'
```

Figure 7.2c. Cumulativity rules (`cumul-Spec0`)

```
| cumul_beta : forall na t b a,
  Σ ;;; Γ ⊢ tApp (tLambda na t b) a ≤s[pb] b {0 := a}
| cumul_iota : forall ci c u args p brs br,
  nth_error brs c = Some br →
  #|args| = (ci.(ci_npar) + context_assumptions br.(bcontext))%nat →
  Σ ;;; Γ ⊢ tCase ci p (mkApps (tConstruct ci.(ci_ind) c u) args) brs  ≤s[pb]
      iota_red ci.(ci_npar) p args br
| cumul_proj : forall p args u arg,
  nth_error args (p.(proj_npars) + p.(proj_arg)) = Some arg →
  Σ ;;; Γ ⊢ tProj (i, pars, narg) (mkApps (tConstruct i 0 u) args) ≤s[pb] arg
```

Figure 7.2d. Computation rules for destructors (`cumulSpec0`)

[TS17]: Timany et al. (2017), *Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC)*

implement cumulativity of inductive types [TS17]. The latter two apply respectively to *fully applied* inductive types and inductive constructors, that can be considered equal if their arguments are one-to-one convertible, and their universe levels are correctly related. This means that *e.g.* the `nil` constructor of polymorphic lists *always* satisfies that `nil@{u} A` is convertible to `nil@{u'} A`, irrespective of the universe levels `u` and `u'`.

Last are the rules for computation. The three rules of Figure 7.2d are for destructors, *i.e.* applied functions, pattern-matching on a constructor, and projections. The β rule for functions directly uses substitution: `b{0 := a}` denotes the substitution of `a` for the variable of De Bruijn index `0` in `b`. Similarly, the `iota_red` function computes the substitution of the branch `br` by the arguments of the constructor `args`. Finally, the rule for projections simply selects the right field of the record.

The next three rules deal with definitions (Figure 7.2e). Rule `cumul_zeta` directly reduces a let-binder into a substitution, while definitions are unfolded using `cumul_rel` and `cumul_delta`, respectively those of the local context or the global environment. In the latter case, the definition must be instantiated with a universe instance,[6] which is denoted `@[u]`.

6: A list of universe levels, corresponding to its polymorphic universe variables.

The last rules (Figure 7.2f) pertain to the reduction of (co-)fixed-points. In all cases, they are unfolded in a guarded fashion, in order to avoid a non-terminating behaviour. On fixed-points, this guard is that they have to be applied to a constructor, and dually co-fixed-points are unfolded when they are forced, either by a pattern-matching or a projection. In both cases, this

```
| cumul_zeta : forall na b t b',
  Σ ;;; Γ ⊢ tLetIn na b t b' ≤s[pb] b' {0 := b}
| cumul_rel i body :
  option_map decl_body (nth_error Γ i) = Some (Some body) →
  Σ ;;; Γ ⊢ tRel i ≤s[pb] lift0 (S i) body
| cumul_delta : forall c decl body (isdecl : declared_constant Σ c decl) u,
  decl.(cst_body) = Some body →
  Σ ;;; Γ ⊢ tConst c u ≤s[pb] body@[u]
```

Figure 7.2e. Computation rules for definitions (`cumulSpec0`)

```
(** Fix unfolding, with guard *)
| cumul_fix : forall mfix idx args narg fn,
  unfold_fix mfix idx = Some (narg, fn) →
  is_constructor narg args = true →
  Σ ;;; Γ ⊢ mkApps (tFix mfix idx) args ≤s[pb] mkApps fn args
| cumul_cofix_case : forall ip p mfix idx args narg fn brs,
  unfold_cofix mfix idx = Some (narg, fn) →
  Σ ;;; Γ ⊢ tCase ip p (mkApps (tCoFix mfix idx) args) brs
     ≤s[pb] tCase ip p (mkApps fn args) brs
| cumul_cofix_proj : forall p mfix idx args narg fn,
  unfold_cofix mfix idx = Some (narg, fn) →
  Σ ;;; Γ ⊢ tProj p (mkApps (tCoFix mfix idx) args)
```

Figure 7.2f. Computation rules for fixed-points (`cumulSpec0`)

ensures that the unfolded (co-)fixed-point can reduce further, either by consuming the constructor of its recursive argument, or by producing a constructor to be consumed by the destructor that forced the unfolding.

7.1.3. Typing

With the cumulativity relation defined, we can turn to typing, defined in PCUICTyping. Similarly to cumulativity, `typing` is an inductively defined relation `Σ ;;; Γ ⊢ t : T`, relative to a global environment `Σ` and a local context `Γ`. The rules correspond roughly to ones we already went over in Chapter 3.

The first set of typing rules, given in Figure 7.3a, pertain to the purely functional fragment. There are not many differences there with respect to Figure 3.2. Rule `type_Rel` looks up for the type of a variable in the context, and ensures that said context is well-formed: the `wf_local` predicate corresponds to $\vdash \Gamma$, but here as everything else it is relative to a global environment. Rule `type_Sort` uses the `super` function to compute the successor of an algebraic universe, and similarly Rule `type_Prod` uses `sort_of product` to compute the universe level of a Π-type.[7] Context extension with an assumption, *i.e.* a variable without a body, is written `Γ ,, vass na A`, and used as expected in Rules `type_Prod` and `type_Lambda`. Finally, `type_App` is for application. It contains an assumption that the product is well-formed, which is not strictly speaking needed once we prove validity, but is useful in some cases to provide a needed induction hypothesis.

The rule for local definitions, given in Figure 7.3b, is similar to the one for λ-abstractions, with the only difference that the body too is typed, and that the context is extended with a definition, *i.e.* a variable with a body, which is written `Γ ,, vdef na b B`.

7: This not only includes computing the maximum of two algebraic universes expressions, but also handling the impredicativity of the sort Prop.

```
| type_Rel : forall n decl,
  wf_local Σ Γ →
  nth_error Γ n = Some decl →
  Σ ;;; Γ ⊢ tRel n : lift0 (S n) decl.(decl_type)
| type_Sort : forall s,
  wf_local Σ Γ →
  wf_universe Σ s →
  Σ ;;; Γ ⊢ tSort s : tSort (super s)
| type_Prod : forall 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)
| type_Lambda : forall 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
| type_App : forall t na A B s u,
  Σ ;;; Γ ⊢ tProd na A B : tSort s →
  Σ ;;; Γ ⊢ t : tProd na A B →
  Σ ;;; Γ ⊢ u : A →
  Σ ;;; Γ ⊢ tApp t u : B{0 := u}
```

Figure 7.3a. Functional fragment (typing)

```
| type_LetIn : forall 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
```

Figure 7.3b. Local definitions (typing)

Next (Figure 7.3c) are the three rules performing look-ups in the global environment, respectively constants – `type_Const` –, inductive types – `type_Ind` – and inductive constructors – `type_Construct`. In all cases the term should be declared in the global environment, the context well-formed – since these are leaves of a term –, and the universe instance given should respect the constraints coming from the entry in the environment – this is the `consistent\_instance\_ext` predicate.

The rules of Figure 7.3d are the ones for the destructors of (co-)inductive types. Rule `type_Case` is somewhat similar to Rule BoolInd. First, the scrutinee should be of an inductive type, declared in the global environment. Next, the predicate and branches should all be well-typed in the appropriate context – obtained by combining the information stored in `p` or `brs`, with that retrieved from the entry in the environment corresponding to the scrutinee's type. Finally, `case_side_conditions` handles universe instances, checks that the elimination is allowed[8]... The second rule, `type_Proj`, is somewhat similar, albeit a bit simpler: the scrutinee should still be some applied co-inductive/record type, and the type is constructed by substitution from the projection information.

The last typing rules for terms are those for (co-)fixed-points, given in Figure 7.3e. They are almost identical, the main part amounts to checking that the types and bodies of all the mutual definitions are well-typed. The `wf_fixpoint` and `wf_cofixpoint` predicates both check that the definitions are on the same block of mutually-defined (co-)inductive types, and that these are of the right kind – inductive for `tFix` and co-inductive for `tCoFix`. Finally, the `fix_guard` and `cofix_guard` predicates correspond to

8: This is where PCUIC enforces computational irrelevance of proofs, by imposing the so-called "singleton elimination" criterion, which ensures that only inductive types of a certain specific shape – subsingletons – can be matched on to build proof relevant content, so that that content cannot actually depend on the value of a proof.

```
| type_Const : forall cst u decl,
  wf_local Σ Γ →
  declared_constant Σ cst decl →
  consistent_instance_ext Σ decl.(cst_universes) u →
  Σ ;;; Γ ⊢ tConst cst u : decl.(cst_type)@[u]
| type_Ind : forall ind u mdecl idecl,
  wf_local Σ Γ →
  declared_inductive Σ ind mdecl idecl →
  consistent_instance_ext Σ mdecl.(ind_universes) u →
  Σ ;;; Γ ⊢ tInd ind u : idecl.(ind_type)@[u]
| type_Construct : forall ind i u mdecl idecl cdecl,
  wf_local Σ Γ →
  declared_constructor Σ (ind, i) mdecl idecl cdecl →
  consistent_instance_ext Σ mdecl.(ind_universes) u →
  Σ ;;; Γ ⊢ tConstruct ind i u : type_of_constructor mdecl cdecl (ind, i) u
```

Figure 7.3c. Globally defined terms (typing)

```
| type_Case : forall ci p c brs indices ps mdecl idecl,
  let predctx := case_predicate_context ci.(ci_ind) mdecl idecl p in
  let ptm := it_mkLambda_or_LetIn predctx p.(preturn) in
  declared_inductive Σ ci.(ci_ind) mdecl idecl →
  Σ ;;; Γ ,,, predctx ⊢ p.(preturn) : tSort ps →
  Σ ;;; Γ ⊢ c : mkApps (tInd ci.(ci_ind) p.(puinst)) (p.(pparams) ++ indices) →
  case_side_conditions (fun Σ Γ ⇒ wf_local Σ Γ) typing Σ Γ ci p ps
                       mdecl idecl indices predctx  →
  case_branch_typing (fun Σ Γ ⇒ wf_local Σ Γ) typing Σ Γ ci p ps
                     mdecl idecl ptm brs →
  Σ ;;; Γ ⊢ tCase ci p c brs : mkApps ptm (indices ++ [c])
| type_Proj : forall p c u mdecl idecl cdecl pdecl args,
  declared_projection Σ p mdecl idecl cdecl pdecl →
  Σ ;;; Γ ⊢ c : mkApps (tInd p.(proj_ind) u) args →
  #|args| = ind_npars mdecl →
  Σ ;;; Γ ⊢ tProj p c : subst0 (c :: List.rev args) (snd pdecl)@[u]
```

Figure 7.3d. (Co-)inductive destructors (typing)

```
| type_Fix : forall mfix n decl,
  wf_local Σ Γ →
  fix_guard Σ Γ mfix →
  nth_error mfix n = Some decl →
  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 →
  wf_fixpoint Σ mfix →
  Σ ;;; Γ ⊢ tFix mfix n : decl.(dtype)
| type_CoFix : forall mfix n decl,
  wf_local Σ Γ →
  cofix_guard Σ Γ mfix →
  nth_error mfix n = Some decl →
  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 →
  wf_cofixpoint Σ mfix →
  Σ ;;; Γ ⊢ tCoFix mfix n : decl.(dtype)
```

Figure 7.3e. (Co-)fixed-points (typing)

```
| type_Cumul : forall t A B s,
  Σ ;;; Γ ⊢ t : A →
  Σ ;;; Γ ⊢ B : tSort s →
  Σ ;;; Γ ⊢ A ⩽s B →
  Σ ;;; Γ ⊢ t : B
```

Figure 7.3f. Cumulativity (typing)

the guard condition, ensuring that the definitions do not endanger normalization. We come back to those in Section 7.5.

The final rule (Figure 7.3f) is that which uses cumulativity as just defined to change the type of the term, *e.g.* the equivalent of Rule Cum.

From the definition of typing, two more pervasively used definitions follow. We have already encountered wf_local, asserting that a local context is well-formed. Its sibling predicate wf[9] asserts that the global environment is well-formed. It ensures not only that all definitions are properly typed, but also of the validity of various information related to inductive types – in particular the positivity criterion which ensures that inductive definitions are well-founded –, and universe polymorphism.

9: Often replaced by wf_ext, an extension that in addition takes into account the universes of the current definition.

7.2. The Easy Properties: Stabilities

With the main definitions set up, we can turn to the properties that we collectively called stabilities in Section 3.4. These assert that cumulativity and typing as just defined are stable by various ubiquitous operations: extension of the local context[10] and global environment,[11] and substitution, not only for terms,[12] but also for universe variables.[13]

One last property falling in the section of low-hanging fruits as well is the fact that well-typed terms are well-scoped.[14] This well-scoping conditions appears in the transitivity rule for cumulativity (Figure 7.2a) and is a hypothesis for many lemmas.

All of these are proven by induction on the typing derivation. While the proof by themselves are not very surprising, the formalization of the definitions deserves a few comments.

The first point to note is that while the weakening and substitution operations are defined directly by induction on the syntax of terms, the proofs are not done directly on those definitions. Rather, MetaCoq uses notions of renaming and instantiation inspired by the σ-calculus [Aba+91; STS15], as functions from natural numbers to natural numbers for renamings, and to terms for instantiations. Weakening and substitution then correspond respectively to a specific form of renaming and of instantiation, and the stability for the former those follows from more general versions for the latter. For instance, weakening is but a consequence of general stability by (unconditional) renaming, as presented in Property 3.1. This approach makes it easier to handle the complex binding structures present in the syntax of PCUIC, which require parallel substitution or lifting by a whole context at once, operations that are easier to handle in the general framework of the σ-calculus.

More interestingly and novel, the same approach is taken also for well-scoping: MetaCoq generalizes that predicate into a way of lifting any boolean

10: PCUICWeakeningConv and PCUICWeakeningTyp.

11: PCUICWeakeningEnvConv and PCUICWeakeningEnvTyp.

12: PCUICInstConv and PCUICSubstitution

13: PCUICUnivSubstitutionConv and PCUICUnivSubstitutionTyp

14: PCUICClosedTyp.

[Aba+91]: Abadi et al. (1991), *Explicit substitutions*

[STS15]: Schäfer et al. (2015), *Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions*

function on natural numbers – seen as a property of variables – to a boolean function on terms.[15] This makes it easy to relate the σ-calculus to well-scoping assumptions, *e.g.* to show that if two substitutions σ and σ' agree on the free variables of a term t, then the application of σ and σ' to t are equal.

15: This is on_free_vars.

```
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).
```

Figure 7.4. Well-formed substitution (subslet)

A last point of interest is the definition of a well-formed substitution, a predicate called subslet. Indeed, the usual typing judgment for substitutions is of the form $\Delta \vdash \sigma : \Gamma$, meaning that σ maps each assumption $(x : A) \in \Gamma$ to a term t such that $\Delta \vdash t : T[\sigma]$. But in our setting we must account for variables that can be *defined* in Δ. This leads to the definition of a well-formed substitution as in Figure 7.4. A similar definition, called well_subst, is also available for instantiations.

7.3. Things Get Serious: Confluence

```
Inductive red1 (Σ : global_env) (Γ : context) : term → term → Type :=
(** Reductions *)
| red_beta na t b a :
  Σ ;;; Γ ⊢ tApp (tLambda na t b) a ⇝ b {0 := a}

| red_zeta na b t b' :
  Σ ;;; Γ ⊢ tLetIn na b t b' ⇝ b' {0 := b}
…
(** Congruences*)
| app_red_l M1 N1 M2 : Σ ;;; Γ ⊢ M1 ⇝ N1 → Σ ;;; Γ ⊢ tApp M1 M2 ⇝ tApp N1 M2
| app_red_r M2 N2 M1 : Σ ;;; Γ ⊢ M2 ⇝ N2 → Σ ;;; Γ ⊢ tApp M1 M2 ⇝ tApp M1 N2
…
where " Σ ;;; Γ ⊢ t ⇝ u " := (red1 Σ Γ t u).

Definition red Σ Γ := clos_refl_trans (fun t u : term ⇒ Σ;;; Γ ⊢ t ⇝ u).
```

Figure 7.5. One-step reduction and reduction

As in Section 3.4, the next step is to define reduction and establish its properties. An excerpt of the definitions is given in Figure 7.5: reduction is red, defined as as the reflexive, transitive closure of one-step reduction red1. The former is written Σ ;;; Γ ⊢ t ⇝* u, and the latter Σ ;;; Γ ⊢ t ⇝ u.

7.3.1. Parallel reduction

The proof of confluence follows the standard Tait-Martin-Löf approach as exposed by Takahashi [Tak95]. It relies on a notion of *parallel reduction* $\Rightarrow$, which can reduce multiple redexes present in a term t in parallel. As an example, the rules for application are given in Figure 7.6. The generic congruence rule allows reduction to happen in parallel in both the function and argument. Moreover, if the function is an abstraction, a β step can also be fired simultaneously with those. Note that this does *not* allow reducing further redexes that would be produced by the substitution. For instance, we do not have

$$(\lambda\, f{:}\,\mathbf{N} \to \mathbf{N}.\, f\ 0)\ (\lambda\, x{:}\,\mathbf{N}.x) \Rightarrow 0$$

because the redex $(\lambda\, x{:}\,\mathbf{N}.x)\ 0$ only appears after a first step of substitution.

[Tak95]: Takahashi (1995), *Parallel Reductions in λ-Calculus*

$$\frac{t \Rightarrow t' \qquad u \Rightarrow u'}{t\ u \Rightarrow t'\ u'}$$

$$\frac{t \Rightarrow t' \qquad u \Rightarrow u'}{(\lambda\, x{:}\,A.\, t)\ u \Rightarrow t'[x := u']}$$

Figure 7.6. Parallel reduction for application

Parallel reduction has two interesting properties. First, it is related to standard reduction.

Lemma 7.1. Parallel reduction and reduction

We have $\to^1\ \subset \Rightarrow\ \subset \to^\star$.

This implies that if parallel reduction is confluent, then so is reduction.

But the interesting characteristic of parallel reduction, which reduction does not satisfy, is the *diamond property*, a strong version of local confluence,[16] which contrarily to it implies confluence even in the absence of normalization. Thanks to Lemma 7.1 above, in order to establish confluence of reduction, it suffices to show this diamond property.

16: Which says that if $t \to^1 t_1$ and $t \to^1 t_2$ then there exists t' such that $t_1 \to^\star t'$ and $t_2 \to^\star t'$.

The proof idea goes as follows. First, show that parallel reduction is substitutive, in the following sense.

Lemma 7.2. Parallel reduction is substitutive

If $t \Rightarrow t'$ and $u \Rightarrow u'$ then $t[x := u] \Rightarrow t'[x := u']$.

This allows to define a *best parallel reduct* ρ, which reduces *all* possible redexes in parallel,[17] and to show that it is really a best reduct.

17: For instance $\rho\,((\lambda\, x{:}\,A.\ t)\ u)$ is $\rho\,(t)[x := \rho\,(u)]$.

Lemma 7.3. Triangle property

Given a term t, if $t \Rightarrow t'$ then $t' \Rightarrow \rho\,(t)$.

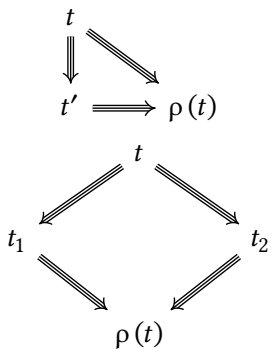

Figure 7.7. The triangle and diamond properties, as diagrams

This is enough to get the diamond property, because any two parallel reducts of a term t both reduce to $\rho\,(t)$ *in one step*. This basically amounts to firing in both reducts all the redexes that could have been triggered but have not been.

Lemma 7.4. Diamond property

Given a term t and t_1, t_2 such that $t \Rightarrow t_1$ and $t \Rightarrow t_2$, there exists t' such that $t_1 \Rightarrow t'$ and $t_2 \Rightarrow t'$.

From this, it follows by a bit of diagram chasing that parallel reduction is confluent, and thus that reduction is.

Theorem 7.5. Confluence of reduction

> Reduction is confluent, that is if $t \to^\star t_1$ and $t \to^\star t_2$ then there exists t' such that $t_1 \to^\star t'$ and $t_2 \to^\star t'$.

Note that it can directly be shown without resorting to parallel reduction that reduction is *locally* confluent. However, in the absence of normalization to apply Newman's lemma, this is not enough to ensure confluence. But, while we can hope that normalization holds for well-typed term, it is clearly false on untyped terms. Yet, it is crucial to get confluence on those, as otherwise we would not be able to establish injectivity of type constructors and thus subject reduction with our untyped notion of cumulativity. Thus, this detour through parallel reduction is really unavoidable.

7.3.2. Formalizing Takahashi's proof

The previous section sets down a quite precise plan, that we can almost directly follow in MetaCoq. There is one important subtlety though: because of local definitions, reduction depends on contexts, so these must be taken into account in parallel reduction. But bodies of definitions should also be reduced by parallel reduction, and so the actual relation is between pairs of a context and a term, something like $\Gamma, t \Rrightarrow \Gamma', t'$.

Apart from this difficulty[18] the plan can be followed quite closely. Parallel reduction is defined as `pred1` in PCUICParallelReduction. Then the diamond property is proven in PCUICParallelReductionConfluence – ρ is `rho`, and the diamond property itself is `pred1_diamond`. Finally, PCUICConfluence goes back from this to properties of reduction, concluding with its confluence (`red_confluence`).

18: And the technicality of defining ρ in a terminating fashion, which is done using the Equations plugin [SM19].

[SM19]: Sozeau et al. (2019), *Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq*

7.4. Reaping the Fruits: the Road to Subject Reduction

With confluence proven, it is time to reap the fruits. First, declarative cumulativity, as used to define typing, can be related to algorithmic cumulativity. This entails many useful consequences, including injectivity of type constructors. A series of important properties then follow, culminating with subject reduction.

7.4.1. Algorithmic cumulativity

The first use of confluence is to relate declarative cumulativity as used to define typing to algorithmic cumulativity – `cumulAlgo` – defined as reduction to terms related by the α-pre-order $\leq_\alpha$.[19] But in the setting of PCUIC this relation, called `leq_term` in the formalization, is far from being trivial to define! It needs to handle algebraic universe levels, but also polymorphic inductive types. Moreover, it is parameterized over the relation used to compare universes, so that it can also be used to express "pure" α-equality[20] when instantiated with equality rather than universe comparison.

19: The generalization of α-equality to handle cumulativity.

20: Meaning that the terms can differ only on binder names, but that their universe must be syntactically the same rather than related using the constraints present in the environment.

To show that algorithmic conversion is equivalent to declarative conversion, confluence is the major ingredient, but it is not enough. Instead, we also need to show that reduction interacts well with this α-pre-order.

Lemma 7.6. The α-pre-order is a simulation (`red1_eq_term_upto_univ_l`)

> If $t \leq_\alpha t'$ and $\Gamma \vdash t \to^1 u$ then there exists some u' such that $\Gamma \vdash t' \to^1 u'$ and $u \leq_\alpha u'$.

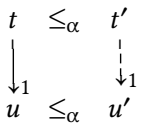

Figure 7.8. Simulation, as a diagram

While the proof is still relatively straightforward since t and t' have the same structure, it becomes much more challenging if we wish to integrate extensionality rules such as η-equality or strict propositions[21] into $\leq_\alpha$.

21: Two features present in Coq but not yet in MetaCoq, exactly due to this kind of difficulties.

Combining this simulation property with confluence, transitivity of algorithmic cumulativity follows, and finally its equivalence with declarative cumulativity.

Theorem 7.7. Equivalence of the presentations of cumulativity

> Algorithmic and declarative cumulativity are equivalent.

This equivalence is the main theorem of PCUICConversion – one direction is `cumulSpec_cumulAlgo`, and the other is `cumulAlgo_cumulSpec`. That central file also proves multiple variants of injectivity of type constructors. For instance, injectivity of function types is `ws_cumul_pb_Prod_Prod_inv`, and the stronger form used in completeness of bidirectional typing (Lemma 4.2) is `ws_cumul_pb_Prod_r_inv`.

7.4.2. Reaping the fruits

With injectivity settled, we can get to the main properties of typing. The easiest is `validity`, asserting that if $\Gamma \vdash t : T$ then T is a well-formed type in Γ. The second is that typing is insensible to names (`typing_alpha`): if two terms differ only in variable names and one is typable, then so is the other. And, finally, comes `subject_reduction`.

While the main proofs of these theorems are far from simple,[22] an important part of the proof effort is actually required ahead of them in PCUICInductives and PCUICInductiveInversion, in order to show that the various contexts, substitutions, applications, etc. that appear in conversion and typing for inductive types behave as expected.

22: That for subject reduction needs more than 1500 lines for the main induction!

Regarding subject reduction, a caveat applies. Because the "positive" presentation of co-fixed-points does not preserve types, as explained in Section 3.6.6, only the "negative" presentation based on projections is allowed. In practice, this means that the typing rule `type_case` of Figure 7.3d forbids the scrutinee from being of a co-inductive type.[23] Hence, while reduction and conversion take this presentation into account,[24] thus showing at least that it is confluent, it can never appear in a well-typed term.

23: This is part of the `case_side_conditions`.

24: See the `cumul_cofix_case` constructor in Figure 7.2f.

An alternative solution, which would allow an easier transition away from today's positive co-inductive types, is to see co-inductive scrutinees as effectful terms and restricts predicates allowed for dependent elimination to

be linear, following Pédrot and Tabareau [PT17]. This approach is not formalized in MetaCoq – yet –, but the project provides a natural setting to explore this kind of questions in all their gritty details before working towards an actual implementation in Coq.

[PT17]: Pédrot et al. (2017), *An Effectful Way to Eliminate Addiction to Dependence*

7.5. Gödel's Thorn in the Side: Normalization

One last important property remains: normalization. In PCUIC, the key constraint to ensure it is the guard condition, to which (co-)fixed-points are subject in order to ensure that they are well-founded – see Section 3.6.5. However, due to Gödel's second incompleteness theorem, if PCUIC is consistent then it cannot prove its own consistency. But if normalization were provable, then so would be consistency. There is therefore no hope to give a guard condition and prove that it entails normalization.

7.5.1. An abstract guard condition

```
Inductive FixCoFix : Type := Fix | CoFix.

Axiom guard : FixCoFix → global_env_ext → context
      → mfixpoint term → Prop.

Definition fix_guard := guard Fix.
Definition cofix_guard := guard CoFix.
```

Figure 7.9. The `guard` axiom

Instead, MetaCoq takes a different approach. The existence of a guard condition is *assumed* in an abstract, axiomatic fashion – see Figure 7.9 – and used in typing – see Figure 7.3e. Similarly, PCUICGuardCondition assumes properties of this axiomatic guard: it should be stable by universe and term substitution, extension of the global environment, cumulativity of the local context insensible to names, and, most importantly, stable by reduction.

These abstract properties are enough to handle the whole development outlined above. In other words, given any notion of guard condition that satisfies the criteria of PCUICGuardCondition, typing satisfies injectivity of type constructors, validity, subject reduction... Thus, our abstract approach provides a precise characterization of the properties the guard condition needs to satisfy in order for typing to be well-behaved.

In particular, since the trivial guard condition that is always true fulfils the requirements, none of the aforementioned properties rely on normalization – or consistency of the theory. Thus, PCUIC is a safe programming language, *unconditionally*.

7.5.2. The normalization axiom

But of course we need more. In particular, if we wish to define a convertibility check inside Coq, which only allows to define terminating functions, we must know that reduction is terminating. Once again, we axiomatize the necessary axiom of (strong) normalization, as given in Figure 7.10. Note

```
Axiom normalisation :
wf_ext Σ →
forall Γ t,
  welltyped Σ Γ t →
  Acc (cored Σ Γ) t.
```

Figure 7.10. The `normalization` axiom

that this axiom takes exactly the form we gave to normalization in Property 3.14, as the accessibility of any well-typed term for co-reduction. This is done under the assumption that the global context is well-formed, otherwise it could contain *e.g.* non-positive inductive types which could be used to define non-terminating terms.

Using this axiom, it is possible to prove consistency of PCUIC, as shown in PCUICConsistency. The rough idea of the proof is that given for Property 3.15, *i.e.* to deduce consistency from canonicity. However, instead of proving progress directly we rely on a proven-complete function computing weak-head normal forms implemented as part of the type-checker.[25] Assuming an inhabitant `t` of $\Pi b : \mathrm{Prop}.\ b$[26] in any axiom-free global environment Σ, the proof extends Σ with the empty inductive type $\bot$, use `t` to construct a weak-head normal inhabitant of that type, and from this finally derive a contradiction.

25: Which can be seen as a constructive witness of canonicity!

26: Corresponding to the MetaCoq term `tProd b (tSort Prop_univ) (tRel 0)`.

8. Building a Certified Kernel

With the meta-theory set down, we can turn to building a kernel – and proving that it is correct. The first step (Section 8.1) is to move from the declarative specification of Chapter 7 to a bidirectional presentation, closer to the kernel we wish to implement. Once this specification is set down, we can get to the kernel itself. Section 8.2 goes over the implementation of the global environment and the cumulativity check, and Section 8.3 describes the type-checker. Finally, Section 8.4 describes two extra functions belonging to the safe kernel: re-typing, and checking of global environment.

I personally contributed the formalizations of Section 8.1, the completeness part of Section 8.3 – by modifying the pre-existing proven-correct type-checker –, and heavily modified re-typing – part of Section 8.4.

8.1 Bidirectional Typing, Formalized 97
8.1.1 Definitions 97
8.1.2 Equivalence with undirected typing 98
8.1.3 Properties of bidirectional typing 99
8.2 Before Typing 99
8.2.1 Abstract environment 100
8.2.2 Cumulativity checking 100
8.3 Correct and Complete Inference 101
8.4 Beyond Typing: Environment Checking and Re-Typing 104
8.4.1 Re-Typing 104
8.4.2 Environment Checking 104

8.1. Formalized Bidirectional Typing

We already saw the main theoretical ideas around our approach to bidirectional typing in Part 'Bidirectional Calculus of Inductive Constructions', so let us get to their implementation in the formalization.

8.1.1. Definitions

Before we can get to the definition of typing, we must go through the small BDEnvironmentTyping, which is dedicated to refining a few definitions on contexts in the bidirectional setting. First, in the case of a definition `Γ ,, vdef na b T`, `wf_local` enforces that `T` is a well-formed type, and that `b` has type `T`. In the bidirectional setting we want to use constrained inference $\triangleright_{\square}$ for the first, and checking for the second, but the generic definition on which `wf_local` is built[1] only allows for a single parameter – instantiated with `typing` in the case of `wf_local`. Similarly, we need a definition expressing that a context Δ is well-formed *over another context* Γ, but which does not enforce Γ to be well-formed *a priori* – *e.g.* something more precise than simply $\vdash \Gamma, \Delta$. This allows to stay faithful to McBride's discipline[2] when typing context extensions, by only demanding that the extension is well-formed, but not the initial segment.[3]

The bidirectional typing judgment is defined in BDTyping, as a set mutual defined inductive predicates: one for inference, one for checking, and one for each constrained inference, *e.g.* respectively sorts, Π-types and inductive types. The definition of the predicates themselves is very close to that of Figures 4.2a and 4.2b for the functional fragment, the main innovation being that constrained inference – written `Σ ;;; Γ ⊢ t ▷Π (na,A,B)` for Π-types – takes the variable name, domain and codomain of the inductive type as three separate arguments, which our pen-and-paper notation did not make explicit. The predicates defined in BDEnvironmentTyping are used in the definition of inference for the `tCase` node, where we want to ensure that the context extensions used to type the predicate and branches are well-formed.

1: Called `All_local_env`.

2: Taken from McBride [McB18], see its exposition in Section 4.1.1.

[McB18]: McBride (2018), *Basics of Bidirectionalism*

3: Which is an input, and thus should not be re-checked.

Regarding the notions of computation, the definitions of constrained inference use full reduction rather than weak-head reduction, mainly because MetaCoq currently lacks a treatment of the latter adequate for our needs.[4] As for cumulativity, the algorithmic variant is used in the checking rule, but this is relatively irrelevant, since the equivalence between both presentations of cumulativity appears much earlier in the development than bidirectional typing.

4: In particular, a standardization theorem is missing, which would be needed to show the analogue of Lemma 4.2 for weak-head reduction.

Maybe more interesting from the formalization point of view is how we obtain a usable induction principle. This is a common issue in MetaCoq: while Coq is able to detect that our inductive definitions are well-founded, the default generation is often unable to derive a sensible induction principle, and neither are the `Scheme` specialized commands. This is due to their nested character, *i.e.* the presence of lists and records containing recursive instances of the inductive types as arguments to the type constructors. The bidirectional typing predicate is the paroxysmal example of this, as it reaches the limit of expressiveness offered by Coq's inductive types: it is not only nested, but also mutual. We thus have to prove our desired induction principle by hand. To do so, we introduce a notion of "generic" typing object `typing_sum`, together with a notion of size for such a typing object, and finally show the induction principle `bidir_ind_env` by well-founded induction on that size.

This induction principle is not as strong as we might expect, as it does not provide the extra induction hypothesis on inputs that would go with McBride's discipline. Ideally, we could use this discipline in order to thread the well-formation invariants, giving stronger induction hypotheses. I did not try to take this path and prove such a strong induction principle, as it did not seem so easy: it would effectively correspond to an inline proof of validity. Instead, the discipline is reflected in the choice of the predicates proven by induction. For instance, in the case of correctness, the mutually proven predicate for inference is `wf_local Σ Γ → Σ ;;; Γ ⊢ t : T`, and more generally assumptions are added as pre-condition for all inputs. Still, I conjecture that such a strong induction principle should be provable, if the need would arise, and might be nice in order to factor proofs, by showing once and for all that the rules follow the discipline correctly.

8.1.2. Equivalence with undirected typing

Correctness[5] is shown in BDToPCUIC. The main proof is by induction on the derivation, its key point being to show that well-formation invariants are preserved, and in particular that all contexts that are constructed are valid.

5: Bidirectional typing implies undirected typing, akin to Theorem 4.1.

There is one particular difficulty linked to the `tCase` constructor, and the question of its representation evoked at the very end of Section 5.2. More precisely, the issue is related to the fact that case nodes store the universe instance and parameters of the inductive type being matched upon, in order to be able to construct the context in which the predicate and branches are typed. In undirected typing, the hypothesis on the scrutinee is that it should be of some type

```
mkApps (tInd ci.(ci_ind) p.(puinst)) (p.(pparams) ++ indices)
```

where `ci.(ci_ind)`, `p.(puinst)` and `p.(pparams)` are respectively the inductive type being matched upon, its universe instance, and its parameters – all stored in the case node –, and `indices` are free. From the point of view of bidirectional typing, this rule is invalid:[6] because `indices` is free, this cannot be turned into a checking premise, but it also cannot be directly turned into inference, or even constrained inference, because it is not free enough due to the presence of `p.(pparams)` and `p.(puinst)`. The solution is still to turn it into an inference premise `Σ ;;; Γ ⊢ c ▷{ci} (u,args)`, and to compare the inferred universe instance `u` and parameters – the first part of the list `args` – to those stored in the node, *e.g.* `p.(puinst)` and `p.(pparams)`. But it requires some work to show that this relation between the two lists of parameters is enough to use the second part of `args`, the inferred indices, in place of `indices` above.

6: It is not mode correct [DK21].

[DK21]: Dunfield et al. (2021), *Bidirectional Typing*

In the opposite direction, completeness[7] is also proven by induction, once we have used the injectivity properties of PCUICConversion to show that inference of a type related by cumulativity to a sort, Π- or inductive type implies constrained inference of the corresponding kind. In order to simplify proofs in the case of projections, correctness is used in conjunction with validity, but this could probably be avoided, making the two proofs independent.

7: Undirected typing implies bidirectional typing, akin to Theorem 4.3.

8.1.3. Properties of bidirectional typing

As we did in Theorem 4.4, we show that two inferred types have a common reduct in BDUnique. While the proof requires some playing with well-scoping predicates,[8] it is conceptually *much* simpler than the direct proof of PCUICPrincipality, which shows the existence of principal types without going through bidirectional typing. Indeed, due to the difficulty of the proof, for quite some time only a weaker version was proven. This version that if T and T' are both types for the same term t then there exists a third T'' which is both a type for t and smaller than T and T' for cumulativity.

8: To relate the reduction used to defined constrained inference and the one on which most lemmas around confluence are stated, which is defined directly on well-scoped terms.

Finally, BDStrengthening shows strengthening. Its first important property is that if $\Gamma \vdash t \triangleright T$ then T can only use variables appearing in either t or in the types of Γ (`infering_on_free_vars`), which is *not* true in general in undirected typing.[9] It then goes on with the proof that bidirectional typing is stable under any renaming, while PCUICRenameTyp only shows stability of undirected typing under unconditional renaming. Finally, we get to the proof of `strengthening` *per se*, once we have shown that strengthening is indeed a well-formed renaming.

9: Consider for instance $n : \mathbb{N} \vdash 0 : (\lambda\, x : \mathbb{N}.\ \mathbb{N})\ n : n$ appears in the type, but neither in the body of the term nor any types in the context.

8.2. Before Typing: Environment Querying and Cumulativity Checking

Before we can get to typing, we need to have a look at its two main subroutines: querying the global environment, and cumulativity check.

8.2.1. Abstract environment

The type and cumulativity checking algorithms both need to query the global environment, for two main purposes: retrieving information about previous definitions of constants and inductive types, and checking that (in)equalities between universe expression hold.

While this might seem anecdotal, a surprisingly important amount of time is spent in the actual checker on the second problem, which requires a form of shortest-path algorithm on a graph obtained from the universe constraints, in order to detect the presence of negative cycles. These correspond to violations of the universe stratification. MetaCoq implements such an algorithm, with a proof that it is correct and complete, meaning that the algorithm answers "yes" exactly when there is a mapping from universe levels to integers satisfying all constraints declared in the environment. More details on this can be found in Sozeau et al. [Soz+20, Section 3.3].

[Soz+20]: Sozeau et al. (2020), *Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq*

Sadly, said algorithm is too naive to be actually run on reasonable examples: it is currently the main performance bottleneck of the extracted type-checker. Similarly, the representation of the global environment as a list of definitions is too naive to allow for efficient lookups – Coq uses hash maps instead. While we hope to replace that naive implementation with a more efficient but still certified one, for the moment it is convenient to be able to plug an uncertified but efficient implementation into the extracted type-checker. To do so, we rely on abstract interfaces for the global environment, containing all the possible queries we need to perform, presented in PCUICWfEnv. The naive implementation is shown to be a valid implementation of that interface in PCUICWfEnvImpl.

8.2.2. Cumulativity checking

The most important sub-routine of the type-checker is the test of cumulativity between two terms. The naive way to perform this, since we assume normalization, would be to brutally normalize terms, and compare normal forms up to `leq_term`.[10] But this strategy does not scale as soon as definitions are present, because it eagerly unfolds all of these, resulting in a very inefficient test. MetaCoq implements a more practical strategy, which coarsely does the following:

10: The extension of α-equality to handle cumulativity.

1. reduce both terms being compared to weak-head normal form *without unfolding any definition*;
2. if the two heads match, recursively compare sub-terms;
3. if the two heads do not match, or if the recursive sub-term comparison failed, check if an unfolding is possible which would unblock one of the terms, and if yes, unfold it and go back to the first step.

This means that the cumulativity test must itself resort to a weak-head reduction function.

The difficulty with those functions is that they do not operate by a simple structural induction on terms. Rather, they are defined using a complex abstract machine, operating on terms decomposed into a sub-term and a stack. The termination of that abstract machine is shown using a dependent

lexicographic pre-order, which handles both the well-founded reduction order given by normalization, a structural order on sub-terms and stacks corresponding to a given term, and the different phases of the algorithm. A detailed description of this algorithm and its formalization[11] is given by Théo Winterhalter, who implemented it,[12] in his PhD thesis [Win20, Chapters 21-24].

11: In files PCUICSafeReduce for the implementation of weak-head reduction, and PCUICSafeConversion for the cumulativity checker.

12: It was only proven correct at the time. Jakob Botch Nielsen wrote most of the completeness proof.

[Win20]: Winterhalter (2020), *Formalisation and meta-theory of type theory*

An interesting point that the test of cumulativity and that of typing have in common, is the way they handle their propositional content. First, because we want to avoid issues linked with proof-irrelevance, in most of the formalization definitions are in `Type`, including the reduction, conversion and typing relation. But in the verified kernel we want to enforce the separation between propositional and relevant content. Thus, we use explicit squashing — written $\|T\|$ – to cast a type into a proposition.[13] The main elimination of propositional content into the relevant world we rely on that of accessibility, so that we can define reduction and cumulativity by well-founded induction. As customary in dependently typed code, we also use elimination of falsity in inaccessible branches.

13: Technically, $\|T\|$ is defined as a record of type Prop with a single field of type T.

Second, we write code using the Equations plugin, which lets us write the relevant part of the definition in direct style, but to leave proofs to be filled-in using the proof-mode. The definitions are given in monadic style, relying on what looks like the error monad:[14] the cumulativity- and type-checker return a valid output, or an error message. However, since we wish the functions to be correct by constructions, they must also return a proof, either a witness for the positive answer, or a proof of impossibility in the negative case. This means that the bind of the monad must actually perform a proof when re-raising the error, in order to propagate the impossibility witness.[15] At function definition, this is hidden by notations, so it feels like we are actually using a monad, but under the hood proof obligations are generated each time we use a bind.

14: Given a type of errors E, the functor associated with that monad is $T \mapsto T + E$, its unit is the left injection, and its bind $x >>= f$ either propagates x if it is an error, or applies f otherwise.

15: For instance, if we are typing some `tProd na A B` and the call to typing for `A` fails, we must transform a proof that `A` cannot be well-typed into a proof that `tProd na A B` as a whole cannot either.

8.3. Correct and Complete Inference

Given the work already done in Section 8.1, the definition of a type checking algorithm PCUICTypeChecker itself is rather straightforward: it follows closely the structure laid out by the mutually defined bidirectional judgments, and poses no termination issue as cumulativity does, since it operates by induction on the structure of the term. Actually, rather than a type-checker, the main function we define is `infer`, which performs type inference,[16] from which we can easily define type-checking.

16: Which is decidable thanks to our Church-style syntax.

In more details, the function takes as inputs:

1. an abstract environment implementation;
2. a global environment implemented using that implementation;
3. a context, and a squashed proof that it is well-formed;
4. a term

and it returns either a type and a (squashed) proof that the term infers that type, or an error and a proof that the term cannot infer any type, using the inductive type presented in Figure 8.1. Thus, the function is correct and complete by construction. In fact, we cannot separate the definition from

```
Inductive typing_result_comp (A : Type) : Type :=
  | Checked_comp : A → typing_result_comp A
  | TypeError_comp : type_error → (A → False) →
      typing_result_comp A.
```

Figure 8.1. The "error monad" used for `infer`'s return type

the correctness proof, since the conversion checker expects a well-typed term as input in order to be terminating when it is called.

Figure 8.2 gives – an excerpt of – the algorithm. For a variable `tRel n`, it checks that the variable is bound in Γ, returns its type when it is, and fails otherwise. In the case of a sort `tSort u`, it checks that the universe is well-formed in the current environment, and returns a sort at the next level when it is. In that of a dependent function type `tProd na A B`, it computes the sort of `A` and `B` – in the context extended by `na:A` – using the `infer_type` sub-routine,[17] and builds from those the sort of the product using `sort_of_product`. Functions are similar. The cases of `tLetIn` and `tApp` clearly show the bidirectional structure. For instance, in `tApp t u`, one needs to infer the type `ty` of `t`, then reduce it to some `tProd na A B` using the `reduce_to_prod` function, and finally check that `u` has type `A`. All underscores _ in the terms denote proof obligations, that are filled later on in tactic mode. Although they are hidden, the monadic notations `;;` and `raise` also contain underscores for the propagation of completeness information.[18]

17: Defined beforehand in an "open recursion" flavour, *e.g.* as a function taking `infer` as argument.

18: For instance, `raise e` is syntactic sugar for `TypeError_comp e _`.

Interestingly, the proofs of completeness use uniqueness of inferred types a lot. To see why, consider *e.g.* the case of an application $t\ u$ where the recursive call succeeds on t – say it infers a product type $\Pi\, x{:}\, A.\ B$ – but the one on u fails – giving us a proof p that $u \triangleleft A$ is absurd. We want to raise an error, and thus need to prove that $t\ u \triangleright T$ for any T is absurd. An inversion on that last hypothesis gives some A' and B' such that $t \triangleright_\Pi \Pi\, x{:}\, A'.\ B'$ and $u \triangleleft A'$. But this second property cannot be directly fed p, because the type against which u checks is different! We thus need to use the two inference judgments and uniqueness to conclude that in fact $A \cong A'$, and thus that $u \triangleleft A$, which this time we can the use to derive a contradiction from p.

```
Equations infer
  (Γ : context)
  (HΓ : forall Σ (wfΣ : abstract_env_ext_rel X Σ), || wf_local Σ Γ ||)
  (t : term)
  : typing_result_comp ({ A : term &
      forall Σ (wfΣ : abstract_env_ext_rel X Σ), || Σ ;;; Γ ⊢ t ▷ A || })
  by struct t :=

  infer Γ HΓ (tRel n)
    with inspect (nth_error Γ n) := {
    | exist (Some c) e ⇒ ret ((lift0 (S n)) (decl_type c); _) ;
    | exist None e ⇒ raise (UnboundRel n)
    } ;

  infer Γ HΓ (tVar n) := raise (UnboundVar n) ;

  infer Γ HΓ (tEvar ev _) := raise (UnboundEvar ev) ;

  infer Γ HΓ (tSort u) with inspect (abstract_env_wf_universeb _ X u) := {
    | exist true _ := ret (tSort (Universe.super u);_) ;
    | exist false _ := raise
        (Msg ("Sort contains an undeclared level " ^ string_of_sort u))
  } ;

  infer Γ HΓ (tProd na A B) :=
    s1 ← infer_type  infer Γ HΓ A ;;
    s2 ← infer_type infer (Γ,,vass na A) _ B ;;
    Checked_comp (tSort (Universe.sort_of_product s1.π1 s2.π1);_) ;

  infer Γ HΓ (tLambda na A t) :=
    infer_type infer Γ HΓ A ;;
    B ← infer (Γ ,, vass na A) _ t ;;
    ret (tProd na A B.π1; _);

  infer Γ HΓ (tLetIn n b b_ty b') :=
    infer_type infer Γ HΓ b_ty ;;
    bdcheck 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; _) ;

  infer Γ HΓ (tApp t u) :=
    ty ← infer Γ HΓ t ;;
    pi ← reduce_to_prod (X_type := X_type) Γ ty.π1 _ ;;
    bdcheck infer Γ HΓ u pi.π2.π1 _ ;;
    ret (subst10 u pi.π2.π2.π1; _) ;

…
```

Figure 8.2. Definition of `infer` (excerpt)

8.4. Beyond Typing: Environment Checking and Re-Typing

There are two more functions defined in MetaCoq that are very close to the type-checker.

8.4.1. Re-Typing

The first, which is defined in PCUICSafeRetyping, aims at computing a type for a term which is known to be well-typed. While this seems tautological, it is not: the aim is to extract relevant content out of propositional one. This is useful in practice in *e.g.* the extraction procedure, which maintains the invariant that it operates on well-typed terms, but at times needs to actually compute types to decide whether terms should be erased.

This is also different from standard inference, because knowing *a priori* that the term under consideration is well-typed allows to skip a lot of checks. For instance, to re-type an application $t\ u$, it suffices to infer a product type $\Pi\, x{:}\, A.\ B$ for t, and to return $B[x := u]$, since we know that u has type A.

In order to be useful, this re-typing procedure needs to compute a principal type, and thus its definition was quite complex prior to the formalization of bidirectional typing, effectively inlining a proof of uniqueness of types. Instead, bidirectional typing simplifies greatly both the definition of re-typing and its proof of correctness, by clarifying its specification: instead of computing a principal type out of any type, the function should compute an unsquashed inferred type out of a squashed one.

8.4.2. Environment Checking

The second thing we need to handle is the verification that a whole global environment is well-formed. While the main thing to check is that all definitions are well-typed, there are quite a few more things to be done: checking universes constraints, that inductive definitions are strictly positive, that the variance information used by universe polymorphism is valid... All these are covered in PCUICSafeChecker.

Bidirectional Elaboration for Gradual Typing

We have already seen in Part 'A Certified Kernel for Coq, in Coq' how the structure of bidirectional typing can help with proofs on CIC/PCUIC. But this is far from being the only advantage of the approach. Indeed, the extra control provided on the conversion rule can be instrumental. In this part, we go over one situation where this is the case: the extension of CIC to incorporate gradual features.

Gradual typing arose as an approach to selectively and soundly relax static type checking by endowing programmers with imprecise static types [ST06; Sie+15]. Optimistically well-typed programs are safeguarded by runtime checks that detect violations of statically-expressed assumptions. A gradual version of typed lambda calculus is flexible enough to embed the untyped lambda calculus [Sie+15]. This means that gradually-typed languages tend to accommodate at least two kinds of effects: non-termination and runtime errors.

[ST06]: Siek et al. (2006), *Gradual Typing for Functional Languages*
[Sie+15]: Siek et al. (2015), *Refined Criteria for Gradual Typing*

Originally formulated in terms of simple types, the extension of gradual typing to a wide variety of typing disciplines has been an extremely active topic of research, both in theory and in practice. As part of this quest towards more sophisticated type disciplines, gradual typing was bound to meet with full-blown dependent types. This encounter saw various premises in a variety of approaches to integrate (some form of) dynamic checking with (some form of) dependent types [Ou+04; WF09; KF10; TT15; LT17; DTT18]. Naturally, the highly-expressive setting of dependent types, in which terms and types are not distinct and computation happens as part of typing, raises a lot of subtle challenges for gradualization.

[Ou+04]: Ou et al. (2004), *Dynamic Typing with Dependent Types*
[WF09]: Wadler et al. (2009), *Well-Typed Programs Can't Be Blamed*
[KF10]: Knowles et al. (2010), *Hybrid type checking*
[TT15]: Tanter et al. (2015), *Gradual Certified Programming in Coq*
[LT17]: Lehmann et al. (2017), *Gradual Refinement Types*
[DTT18]: Dagand et al. (2018), *Foundations of Dependent Interoperability*

Of those challenges, one of the first is the place of computation. In the gradual setting, in order to optimistically compare types, conversion is replaced by consistency, a relation akin to unification. This relation is naturally non-transitive, meaning that the usual, undirected setting is not suited for gradualization.[1] Moreover, the semantics of gradual languages is usually explained through an elaboration phase to a second language, responsible for the runtime checks ensuring safety of evaluation. This elaboration is naturally described in a bidirectional system, which furthermore provides enough constraints on the typing derivation so that replacing conversion with consistency is reasonable. Finally, the identification of the role of reduction for constrained inference clarifies how the latter should be extended to incorporate imprecise types. In fact, I told the story upside down: it is the pressing need for bidirectional typing in the context of gradual typing that led me to its investigation!

1: This is because Rule Conv can be applied any number of times, which is sensible only if these successive application amount to just one.

In this part, we go over a collaboration with Kenji Maillard, Éric Tanter and Nicolas Tabareau to address the challenge of gradualizing a full-blown dependently-typed language: CIC [Len+22]. Chapter 9 gives an overview of the challenges and trade-offs involved in gradual dependent types, culminating with the Fire Triangle of Graduality, which identifies an irreconcilable tension between the properties one should demand of such a type system. It ends with a broad picture of our proposed solution to those difficulties, the *Gradual Calculus of Inductive Constructions* (GCIC). Chapter 10 describes precisely this GCIC, via a relation representing type-based, bidirectional elaboration, which represents my main technical contribution to Lennon-Bertrand et al. [Len+22]. Finally, Chapter 11 gives an overview of the rest of our work in the area: models used to establish properties of the target language of the elaboration procedure, and the thorny question

[Len+22]: Lennon-Bertrand et al. (2022), *Gradualizing the Calculus of Inductive Constructions*

of indexed inductive types and consistent reasoning about gradual programs. Due to their absence of direct relation to bidirectional typing and my lower involvement in their technical development, this chapter does not go into full details, but they are of course present in the publications – either Lennon-Bertrand et al. [Len+22], or Maillard et al. [Mai+22].

[Len+22]: Lennon-Bertrand et al. (2022), *Gradualizing the Calculus of Inductive Constructions*

[Mai+22]: Maillard et al. (2022), *A Reasonably Gradual Type Theory*

Gradual Typing Meets Dependent Types

9.

9.0.1 Smoother development with indexed types 109
9.0.2 Defining general recursive functions 110
9.0.3 Large elimination, gradually . 110
9.0.4 Gradually refining specifications 111
9.0.5 Gradual programs or proofs? . 112
9.0.6 Fundamental trade-offs 112
9.1 Safety and Normalization . 112
9.2 Non-Gradual Approaches . 113
9.2.1 Axioms 113
9.2.2 Exceptions 114
9.3 Gradual Simple Types . . . 115
9.3.1 Static semantics 115
9.3.2 Dynamic semantics 115
9.3.3 Conservativity 116
9.3.4 Gradual guarantees 116
9.3.5 Graduality 117
9.4 Graduality and Dependent Types 119
9.4.1 Unknown term and error type 119
9.4.2 Revisiting safety 119
9.4.3 Relaxing conversion 120
9.4.4 Dealing with neutrals 120
9.4.5 DGG *vs* graduality 120
9.4.6 Observational refinement . . . 121
9.5 Fire Triangle of Graduality 122
9.5.1 Preliminary: regular reduction 122
9.5.2 Gradualizing STLC 122
9.5.3 Gradualizing CIC 123
9.5.4 The Fire Triangle in practice . 124
9.6 GCIC: An Overview 124
9.6.1 Three in one 124
9.6.2 Typing, conversion and bidirectional elaboration . . . 127
9.6.3 Precisions and properties . . . 128

Before diving into what GCIC is about, let me first say what it is not about. The aim is not to put forth a unique design or solution, but rather to explore the space of possibilities. Nor is it about a concrete implementation of gradual CIC and an evaluation of its applicability; these are challenging perspectives of their own, which first require the theoretical landscape to be unveiled. Rather, I believe that studying the gradualization of a full-blown dependent type theory like CIC is in and of itself a valuable scientific endeavour, which is very likely to inform the gradual typing research community in its drive towards supporting ever more challenging typing disciplines.

This being said, we can still highlight some practical motivating scenarios for gradualizing CIC, anticipating what could be achieved in a hypothetical gradual version of *e.g.* Coq.

9.0.1. Smoother development with indexed types

Dependent type systems such as CIC, which underpin languages and proof assistants such as Coq, Agda and Idris, among others, are very powerful system to program in, but at the same time extremely demanding. Mixing programs and their specifications is attractive, but challenging.

Consider the example of the vector type $\mathbf{Ve}\,(A, n)$ as defined in Section 3.5. In Coq, its definition is the following:

```
Inductive vec (A : Type) : ℕ → Type :=
| nil  : vec A 0
| cons : A → forall n : ℕ, vec A n → vec A (S n).
```

Indexing the inductive type by its length allows us to define a *total* `head` function, which can only be applied to non-empty vectors:

```
head : forall A n, vec A (S n) → A
```

Developing functions over such structures can be tricky. For instance, what type should the `filter` function be given?

```
filter : forall A n (p : A → 𝔹), vec A n → vec A …
```

The size of the resulting list depends on how many elements in the list actually match the given predicate `p`! Dealing with this level of intricate specification can (and does) scare programmers away from mixing programs and specifications. The truth is that many libraries, such as the Mathematical Components library [MT21], give up on mixing programs and specifications even for simple structures such as these, which are instead dealt with as ML-like lists with extrinsically-established properties. This tells a lot about the current intricacies of dependently-typed programming.

Instead of avoiding the obstacle altogether, gradual dependent types provide a uniform and flexible mechanism to a tailored adoption of dependencies. For instance, one could give `filter` the following gradual type, which makes use of the *unknown term* ? in an index position:

[MT21]: Mahboubi et al. (2021), *Mathematical Components*

```
filter : forall A n (f : A → 𝔹), vec A n → vec A ?
```

This imprecise type means that uses of `filter` will be optimistically accepted by the type-checker, although subject to associated checks during reduction. For instance,

```
head ℕ ? (filter ℕ 4 even [ 0 ; 1 ; 2 ; 3 ])
```

type-checks, and successfully evaluates to `0`, while

```
head ℕ ? (filter ℕ 2 even [ 1 ; 3 ])
```

type-checks but fails during reduction, upon the discovery that the assumption of non-emptiness of the argument to head is in fact incorrect.

9.0.2. Defining general recursive functions

Another challenge of working in CIC is to convince the type-checker that recursive definitions are well-founded. This can either require tight syntactic restrictions, or sophisticated arguments involving accessibility predicates. At any given stage of a development, one might not be in a position to follow any of these. In such cases, a workaround is to adopt the "fuel" pattern, *i.e.* parametrize a function with a clearly syntactically decreasing argument in order to please the termination checker, and to use an arbitrary initial fuel value. In practice, one sometimes requires a simpler way to unplug termination checking, and for that purpose, many proof assistants support external commands or parameters to deactivate termination checking.[1]

1: For instance `{-# TERMINATING #-}` in Agda or `Unset Guard Checking` in Coq.

Because the use of the *unknown type* ? allows the definition of fixed point combinators [ST06; ETG19], one can use this added expressiveness to bypass termination checking locally. This just means that the external facilities provided by specific proof assistant implementations now become internalized in the language.

[ST06]: Siek et al. (2006), *Gradual Typing for Functional Languages*
[ETG19]: Eremondi et al. (2019), *Approximate Normalization for Gradual Dependent Types*

9.0.3. Large elimination, gradually

One of the argued benefit of dynamically-typed languages, which is accommodated by gradual typing, is the ability to define functions that can return values of different types depending on their inputs, such as the following:[2]

```
Definition foo n m := if (n ?> m) then m + 1 else m ?> 0.
```

2: With `?>` a boolean comparison operator.

In a gradually-typed language, one can give such a function the type `?`, or even `ℕ → ℕ → ?` in order to enforce proper argument types, and remain flexible in the treatment of the returned value. Of course, we know very well that in a dependently-typed language, using large elimination, we can simply give `foo` the dependent type:

```
foo : forall (n m : ℕ), if (n ?> m) then ℕ else 𝔹
```

Lifting the term-level comparison `n ?> m` to the type level is extremely expressive, but hard to work with as well, both for the implementer of the function and its clients. In a gradual, dependently-typed setting, one can explore the whole spectrum of type-level precision for such a function, starting from the least precise to the most precise, for instance:

```
foo : ?
foo : ℕ → ℕ → ?
foo : ℕ → ℕ → if ? then ℕ else ?
foo : forall (n m : ℕ), if (n ?> m) then ℕ else ?
foo : forall (n m : ℕ), if (n ?> m) then ℕ else 𝔹
```

At each stage from top to bottom, there is less flexibility – but more guarantees! – for both the implementer of `foo` and its clients. The gradual guarantee[3] ensures that if the function is actually faithful to the most precise type then giving it any of the less precise types above does not introduce any new failure [Sie+15].

3: One of the important properties we seek in our GCIC.

[Sie+15]: Siek et al. (2015), *Refined Criteria for Gradual Typing*

9.0.4. Gradually refining specifications

Let us come back to the `filter` function from the first example. Its fully-precise type requires appealing to a type-level function that counts the number of elements in the list satisfying the predicate – notice the dependency to the input vector `v`:

```
filter : forall A n (p : A → 𝔹) (v : vec A n),
          vec A (count A n p v)
```

Anticipating the need for this function, a gradual specification could adopt the above signature for `filter` but leave `count` unspecified:

```
Definition count A n (p : A → 𝔹) (v: vec A n) : ℕ := ?.
```

This situation does not affect the behaviour of the program compared to leaving the return type index unknown. More interestingly, one could immediately define the base case, which trivially specifies that there are no matching elements in an empty vector:

```
Definition count A n (p : A → 𝔹) (v : vec A n) : ℕ :=
  match v with
  | nil _ _ ⇒ 0
  | cons _ _ _ ⇒ ?
  end.
```

This slight increment in precision provides a little more static checking, for instance: `head ℕ ? (filter ℕ 4 even [])` does not even type-check, instead of failing during reduction.

Again, the gradual guarantee ensures that such incremental refinements in precision towards the proper fully-precise version do not introduce spurious errors. Note that this is in stark contrast with the use of axioms – which will be discussed in more depth in Section 9.2.1. Indeed, replacing correct code with an axiom can simply break typing! For instance, with the following definitions:

```
Axiom to_be_done : ℕ.
Definition count A n (p : A → 𝔹) (v: vec A n) : ℕ :=
  to_be_done.
```

the definition of `filter` does not type-check any more, as the axiom at the type-level is not convertible to any given value.

9.0.5. Gradual programs or proofs?

When adapting the ideas of gradual typing to a dependent type theory, one might expect to deal with programs rather than proofs. This observation is however misleading: from the point of view of the Curry-Howard correspondence, proofs and programs are intrinsically related, so that gradualizing the latter begs for a gradualization of the former. The examples above illustrate mixed programs and specifications, which naturally also appeal to proofs: dealing with indexed types typically requires exhibiting equality proofs to rewrite terms. Moreover, there are settings in which one must consider computationally-relevant proofs, such as constructive algebra and analysis, homotopy type theory, etc. In such settings, using axioms to bypass unwanted proofs breaks reduction, and because typing requires reduction, the use of axioms can simply prevent typing, as illustrated in the last example.

9.0.6. Fundamental trade-offs

Before exposing a specific approach to gradualizing CIC, there is a need for a general analysis of the properties at stake and tensions that arise when gradualizing a dependent type theory.

Thus, in what follows we start by recalling the two cornerstones properties of progress and normalization, and explain the need to reconsider them carefully in a gradual setting (Section 9.1). Next, we show why two obvious approaches based respectively on axioms (Section 9.2.1), and exceptions (Section 9.2.2) are unsatisfying. We then turn to the gradual approach, recalling its essential properties in the simply-typed setting (Section 9.3), and revisiting them in the context of a dependent type theory (Section 9.4). This finally leads us to establish a fundamental impossibility in the gradualization of CIC, which means that at least one of the desired properties has to be sacrificed (Section 9.5). With all set up, we can finally present our gradual, dependently typed system, GCIC, and its main characteristics (Section 9.6).

9.1. Safety and Normalization, Endangered

In the gradual setting, the two cornerstone properties of CIC exposed in Section 3.4, safety[4] and normalization, must be considered with care.

First, any closed term can be ascribed the unknown type ? and then any other type: for instance, $0 :: ? :: \mathbf{B}$ is a well-typed closed term of type $\mathbf{B}$.[5] However, such a term cannot possibly reduce to either tt or ff, so some concessions must be made with respect to safety – at the very least, the notion of canonical forms must be extended.

Second, normalization is endangered. The quintessential example of non-termination in the untyped lambda calculus is the term Ω, defined as $\delta\ \delta$ where δ is $\lambda\, x.\ (x\ x)$. In the *simply-typed lambda calculus*[6], as in CIC, *self-applications* like $\delta\ \delta$ and $x\ x$ are ill-typed. However, when introducing gradual types, one usually expects to accommodate such idioms, and therefore in a standard gradually-typed calculus such as *GTLC*[7] [ST06], a variant of

4: The combination of progress and preservation.

5: We write $a :: A$ for a type *ascription*, used to "force" the term a to inhabit type A. We define it as syntactic sugar for $(\lambda\, x{:}\, A.\ x)\ a$ [ST06], so $0 :: ? :: \mathbf{B}$ is $(0 :: ?) :: \mathbf{B}$. In other systems, it is taken as a primitive notion [GCT16].

[ST06]: Siek et al. (2006), *Gradual Typing for Functional Languages*

[GCT16]: Garcia et al. (2016), *Abstracting Gradual Typing*

6: Hereafter abbreviated as *STLC*.

7: The gradual counterpart to STLC.

Ω that uses $(\lambda\, x{:}?.\ x\ x)$ as δ is well-typed and diverges – *i.e.* reduces indefinitely. The reason is that the domain type of δ, the unknown type ?, is *consistent* with the type of δ itself, $? \to ?$, meaning that we wish to optimistically accept the application as plausibly valid. But at runtime, nothing prevents reduction from going on forever. Therefore, if one aims at ensuring normalization in a gradual setting, some care must be taken to restrict expressiveness.

9.2. Non-Gradual Approaches

9.2.1. Axioms

Let us first address the elephant in the room: why would one want to gradualize CIC instead of simply postulating an axiom for any term – be it a program or a proof – that one does not feel like providing (yet)?

Indeed, we can augment CIC with a wildcard axiom $\mathrm{ax} : \Pi\, A{:}\,\square.\ A$. The resulting system, called *CIC+ax*, has an obvious practical benefit: we can use $\mathrm{ax}\, A$ as a wildcard whenever we are asked to exhibit an inhabitant of some type A and we do not (yet) want to. This is exactly what admitted definitions are in Coq, for instance, and they do play an important practical role during any Coq development.

However, we cannot use the axiom $\mathrm{ax}\, A$ in any meaningful way *at the type level*. For instance, going back to the examples of Section 9.0.1, one might be tempted to give to the `filter` function on vectors the type

```
forall A n (p : A → 𝔹), vec A n → vec A (ax ℕ)
```

in order to avoid the complications related to specifying the size of the vector produced by `filter`. The problem is that the term:

```
head ℕ (ax ℕ) (filter ℕ 4 even [ 0 ; 1 ; 2 ; 3 ])
```

is ill-typed since the type of the filtering expression, `vec A (ax ℕ)`, is not convertible to `vec A (S (ax ℕ))`, as required by `head ℕ (ax ℕ)` in its domain type.

Thus, the axiomatic approach is not useful for making dependently-typed programming any more pleasing. That is, using axioms goes in total opposition to the gradual guarantee – characteristic of gradual languages [Sie+15] – when it comes to the smoothness of the static-to-dynamic checking spectrum: given a well-typed term, making it "less precise" by using axioms for some sub-terms actually results in programs that do not typecheck or reduce any more.

[Sie+15]: Siek et al. (2015), *Refined Criteria for Gradual Typing*

Because CIC+ax amounts to working in CIC with an initial context extended with ax, this theory satisfies normalization as much as CIC, so conversion remains decidable. However, CIC+ax lacks a satisfying notion of safety, because there is an *infinite* number of *stuck* terms that inhabit any type `A`. For instance, in $\mathbf{B}$, we not only have the normal forms tt, ff, and $\mathrm{ax}\, \mathbf{B}$, but also plenty of terms stuck on an elimination of ax, such as $\mathrm{ax}\, (\mathbf{N} \to \mathbf{B})\ 1$ or $\mathrm{ind}_{\mathbf{N}}(\mathrm{ax}\, \mathbf{N}; P; b_0, b_S)$.

9.2.2. Exceptions

[PT18]: Pédrot et al. (2018), *Failure is Not an Option An Exceptional Type Theory*

Pédrot and Tabareau [PT18] present the exceptional type theory *ExTT*, demonstrating that it is possible to extend a type theory with a wildcard term while enjoying a satisfying notion of safety, which coincides with that of programming languages with exceptions.

ExTT is essentially CIC+raise , that is, it extends CIC with an exceptional term raise_A that can inhabit any type A. But instead of being treated as a computational black box like $\mathrm{ax}\, A$, raise_A is endowed with computational content emulating exceptions in programming languages, which propagate instead of being stuck. For instance, in ExTT the following conversion holds:

$$\mathrm{ind}_{\mathbf{B}}(\mathrm{raise}_{\mathbf{B}}\ ; \mathbf{N}; 0, 1) \cong \mathrm{raise}_{\mathbf{N}}$$

Notably, such exceptions are *call-by-name* exceptions, so one can only discriminate exceptions on positive types – *i.e.* inductive types –, not on negative types – *i.e.* function types. In particular, in ExTT, $\mathrm{raise}_{A\to B}$ reduces to $\lambda\, x{:}A.\ \mathrm{raise}_B$. So raise_A is a normal form of A only if A is a positive type.

ExTT has a number of interesting properties. It is normalizing and safe, taking raise_A into account as usual in programming languages, where exceptions are possible outcomes of computation: the canonical forms of a positive type – *e.g.* $\mathbf{B}$ – are either the constructors of that type – *e.g.* tt and ff –, or raise at that type – *e.g.* $\mathrm{raise}_{\mathbf{B}}$. As a consequence, ExTT does not satisfy full canonicity, but a weaker form of it. In particular, it enjoys (weak) logical consistency: any closed proof of $\bot$ is convertible to $\mathrm{raise}_{\bot}$, which is discriminable at $\bot$. It has been shown that we can still reason soundly in an exceptional type theory, either using a parametricity requirement [PT18], or, more flexibly, a different universe hierarchies [Péd+19].

[PT18]: Pédrot et al. (2018), *Failure is Not an Option An Exceptional Type Theory*

[Péd+19]: Pédrot et al. (2019), *A Reasonably Exceptional Type Theory*

[PT20]: Pédrot et al. (2020), *The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects*

8: That is, a term former such as ind.

It is also important to highlight that this weak form of logical consistency is the *most* one can expect in a theory with effects. Indeed, Pédrot and Tabareau [PT20] have shown that it is not possible to define a type theory with full dependent elimination[8] that has observable effects – of which exceptions are a particular case – and at the same time validates traditional canonicity. Settling for less, as explained in Section 9.2.1 for the axiomatic approach, leads to an infinite number of stuck terms, even in the case of booleans, which contradicts the type safety criterion of gradual languages, which only allows for runtime type errors.

Unfortunately, while ExTT solves the safety issue of the axiomatic approach, it still suffers from the same limitation as the axiomatic approach regarding type-level comparison. Indeed, even though we can use raise to inhabit any type, we cannot use it in any meaningful way at the type level. In such a system, the following term is ill-typed

```
head 𝔹 (raise 𝔹) (filter 𝔹 4 even [ 0 ; 1 ; 2 ; 3 ])
```

as `vec A (raise 𝔹)` is still not convertible to `vec A (S (raise 𝔹))`. The reason is that `raise 𝔹` behaves like an extra constructor of type `𝔹`, so that `S (raise 𝔹)` is itself a normal form, and normal forms with different head constructors – `S` and `raise` – are not convertible.

9.3. Gradual Simple Types

Before going on with our exploration of the fundamental challenges in gradual dependent type theory, let us go over some key concepts and expected properties, in the context of simple types.

9.3.1. Static semantics

Gradually typed languages introduce the *unknown type*, written ?, which is used to indicate the lack of static typing information [ST06]. One can understand such an unknown type as an abstraction of the set of possible types that it stands for [GCT16]. This interpretation provides a naive but natural understanding of the meaning of partially-specified types. For instance $\mathbf{B} \to ?$ denotes the set of all function types with $\mathbf{B}$ as domain. Given imprecise types, a gradual type system relaxes all type predicates and functions in order to optimistically account for occurrences of ?. In a simple type system, the main predicate on types is equality, whose relaxed counterpart is called *consistency*[9], usually written $\sim$. For instance, given a function f of type $\mathbf{B} \to ?$, the expression $(f\ \mathrm{tt}) + 1$ should be well-typed. Indeed, f could *plausibly* return a number, given that its codomain is ?, which is consistent with $\mathbf{N}$.

[ST06]: Siek et al. (2006), *Gradual Typing for Functional Languages*

[GCT16]: Garcia et al. (2016), *Abstracting Gradual Typing*

9: Not to be confused with logical consistency!

Note that there are other ways to consider imprecise types, for instance by restricting the unknown type to denote base types – in which case ? would not be consistent with any function type –, or by only allowing imprecision in certain parts of the syntax of types, such as effects [BGT16], security labels [FT13; TGT18], annotations [TF14], or only at the top-level [BMT10]. Here, we do not consider these specialized approaches, which have benefits and challenges of their own, and stick to the mainstream setting of gradual typing in which the unknown type is consistent with any type and can occur anywhere in the syntax of types.

[BGT16]: Bañados Schwerter et al. (2016), *Gradual type-and-effect systems*

[FT13]: Fennell et al. (2013), *Gradual Security Typing with References*

[TGT18]: Toro et al. (2018), *Type-Driven Gradual Security with References*

[TF14]: Thiemann et al. (2014), *Gradual Typing for Annotated Type Systems*

[BMT10]: Bierman et al. (2010), *Adding Dynamic Types to C#*

9.3.2. Dynamic semantics

Having optimistically relaxed typing based on consistency, a gradual language must detect inconsistencies at runtime if it is to satisfy safety, which therefore has to be formulated in a way that encompasses runtime errors.

For instance, if the function f above returns ff, then an error must be raised to avoid reducing to $\mathrm{ff} + 1$ – a closed stuck term, corresponding to a violation of safety. The traditional approach to do so is to avoid giving a direct reduction semantics to gradual programs, and, instead, to elaborate them to an intermediate language with runtime casts, in which casts between inconsistent types raise *errors*[10] [ST06].

10: We write those err.

[ST06]: Siek et al. (2006), *Gradual Typing for Functional Languages*

In such a language, the notion of canonical form used to phrase progress – and, thus, safety – has to account for these newly introduced errors. Indeed, err_A is now a valid canonical form at type A – at least for some types such as $\mathbf{B}$, since, as we explained in Section 9.2.2, call-by-name errors are not normal forms of function types.

Alternatively – and equivalently from a semantics point of view – one can define reduction of gradual programs directly on gradual typing derivations

augmented with evidence about consistency judgments, and report errors when transitivity of such judgments is unjustified [GCT16]. There are many ways to realize each of these approaches, which vary in terms of efficiency and eagerness of checking [HTF10; TF08; SW10; SGT09; TT20; Bañ+21].

[GCT16]: Garcia et al. (2016), *Abstracting Gradual Typing*

[HTF10]: Herman et al. (2010), *Space-efficient gradual typing*
[TF08]: Tobin-Hochstadt et al. (2008), *The Design and Implementation of Typed Scheme*
[SW10]: Siek et al. (2010), *Threesomes, with and without Blame*
[SGT09]: Siek et al. (2009), *Exploring the Design Space of Higher-Order Casts*
[TT20]: Toro et al. (2020), *Abstracting gradual references*
[Bañ+21]: Bañados Schwerter et al. (2021), *Abstracting Gradual Typing Moving Forward: Precise and Space-Efficient*

9.3.3. Conservativity

A first important property of a gradual language is that it is a *conservative extension* of a related static typing discipline: the gradual and static systems should coincide on static terms. This property is hereafter called *conservativity*, with respect to a given static system. Technically, Siek and Taha [ST06] prove that typing and reduction of GTLC and STLC coincide on their common set of terms – *i.e.* those which are fully precise. An important aspect of conservativity is that the type formation rules and typing rules themselves are also preserved, up to the presence of ? as a new type and the adequate lifting of predicates and functions [GCT16]. While this aspect is often left implicit, it ensures that the gradual type system does not behave in ad hoc ways on imprecise terms.

[ST06]: Siek et al. (2006), *Gradual Typing for Functional Languages*

[GCT16]: Garcia et al. (2016), *Abstracting Gradual Typing*

Note that, despite its many issues, CIC+ax (Section 9.2.1) satisfies conservativity (with respect to CIC): all pure – *i.e.* axiom-free – CIC terms behave as they would in CIC. More precisely, two CIC terms are convertible in CIC+ax if and only if they are convertible in CIC. Importantly, this does not mean that CIC+ax is a conservative extension of CIC *as a logic* – which it clearly is not!

9.3.4. Gradual guarantees

The early accounts of gradual typing emphasized consistency as the central idea. However, Siek et al. [Sie+15] observed that this characterization left too many possibilities for the impact of type information on program behaviour, compared to what was originally intended [ST06]. Consequently, they brought forth type *precision*[11] as the key notion, from which consistency can be derived: two types A and B are consistent if and only if there exists T such that $T \sqsubseteq A$ and $T \sqsubseteq B$. The unknown type ? is the most imprecise type of all, *i.e.* $T \sqsubseteq ?$ for any T. Precision is a pre-order that can be used to capture the intended *monotonicity* of the static-to-dynamic spectrum afforded by gradual typing. The static and dynamic *gradual guarantees* respectively specify that typing and reduction should be *monotone with respect to precision*: losing precision should not introduce new static or dynamic errors. These properties require precision to be extended from types to terms. Siek et al. [Sie+15] present a natural extension that is purely syntactic: a term is more precise than another if they are α-equal, except for their type annotations, which can be more precise in the former.

[Sie+15]: Siek et al. (2015), *Refined Criteria for Gradual Typing*

[ST06]: Siek et al. (2006), *Gradual Typing for Functional Languages*

11: Denoted $\sqsubseteq : A \sqsubseteq B$ means that A is more precise than B, *i.e.* that A contains more static information than B.

The static gradual guarantee (*SGG*) ensures that imprecision does not alter typeability.

Property 9.1. *Static Gradual Guarantee*

> If $t \sqsubseteq u$ and $\vdash t : T$, then $\vdash u : U$ for some U such that $T \sqsubseteq U$.

This SGG captures the intuition that "sprinkling ? over a term" maintains its typeability. As such, the notion of precision $\sqsubseteq$ used to formulate the

SGG is inherently syntactic, over as-yet-untyped terms: typeability is the *consequence* of the SGG theorem.

The dynamic gradual guarantee (*DGG*) is the key result that links the syntactic notion of precision to reduction: if $t \sqsubseteq t'$ and t reduces to some value v, then t' reduces to some value v' such that $v \sqsubseteq v'$; and if t diverges, then so does t'. This entails that $t \sqsubseteq t'$ means that t may error more than t', but otherwise they should behave the same. Instead of the original formulation of the DGG by Siek et al. [Sie+15], New and Ahmed [NA18] appeal to the semantic notion of observational error-approximation to capture the relation between two terms that are contextually equivalent, except that one may fail more:[12]

[NA18]: New et al. (2018), *Graduality from Embedding-Projection Pairs*

12: Observational error-approximation does not mention the case where $\mathcal{C}[t]$ reduces to tt or ff, but the quantification over all contexts ensures that, in that case, $\mathcal{C}[t']$ must reduce to the same value.

Definition 9.2. *Observational error-approximation*

> A term $\Gamma \vdash t : T$ observationally error-approximates a term $\Gamma \vdash t' : T'$, noted $t \preccurlyeq^{\text{ob}} t'$, if for all boolean-valued observation contexts $\mathcal{C} : (\Gamma \vdash T) \Rightarrow (\vdash \mathbf{B})$ closing over all free variables, either
>
> - $\mathcal{C}[t]$ and $\mathcal{C}[t']$ both diverge;
> - otherwise if $\mathcal{C}[t'] \to^\star \text{err}_\mathbf{B}$, then $\mathcal{C}[t] \to^\star \text{err}_\mathbf{B}$.
>
> Two terms t and t' are *observationally equivalent*, written $t \approx^{\text{ob}} t'$, if they are related by observational error-approximation in both directions.

Using this semantic notion, the DGG simply states that term precision implies observational error-approximation:

Property 9.3. *Dynamic Gradual Guarantee*

> If $t \sqsubseteq t'$ then $t \preccurlyeq^{\text{ob}} t'$.

While often implicit, it is important to highlight that the DGG is relative to both the notion of precision $\sqsubseteq$ and the notion of observations $\preccurlyeq^{\text{ob}}$. Indeed, it is possible to study alternative notions of precisions beyond the natural definition stated by Siek et al. [Sie+15]. For instance, following the Abstracting Gradual Typing methodology [GCT16], precision follows from the definition of gradual types through a concretization to sets of static types. This opens the door to justifying alternative precisions, *e.g.* by considering that the unknown type only stands for specific static types, such as base types. Additionally, variants of precision have been studied in more challenging typing disciplines where the natural definition seems incompatible with the DGG, see *e.g.* Igarashi, Sekiyama, and Igarashi [ISI17]. As we will soon see, it can also be necessary in certain situations to consider another notion of observations.

[Sie+15]: Siek et al. (2015), *Refined Criteria for Gradual Typing*

[GCT16]: Garcia et al. (2016), *Abstracting Gradual Typing*

[ISI17]: Igarashi et al. (2017), *On Polymorphic Gradual Typing*

9.3.5. Graduality

As we have seen, the DGG is relative to a notion of precision, but what should this relation be? To go beyond a syntactic axiomatic definition of precision, New and Ahmed [NA18] characterize the good dynamic behaviour of a gradual language: the runtime checking mechanism used to define it,

[NA18]: New et al. (2018), *Graduality from Embedding-Projection Pairs*

such as casting, should only perform type-checking, and not otherwise affect behaviour.

Specifically, they mandate that precision gives rise to *embedding-projection pairs* (*ep-pairs*): the cast induced by two types related by precision forms an adjunction, which induces a retraction. In particular, going to a less precise type and back is the identity: for any term a of type A, and assuming $A \sqsubseteq B$, $a :: B :: A$[13] should be observationally equivalent to a. For instance, $1 :: ? :: \mathbf{N}$ should be equivalent to 1. Dually, when gaining precision, there is the potential for errors: given a term b of type B, $b :: A :: B$ may fail. By considering error as the most precise term, this can be stated as $b :: A :: B \sqsubseteq b$. For instance, with the imprecise successor function f of type $? \to ?$, defined as $\lambda n{:}?.\ \mathsf{S}(n) :: ?$, we have $f :: \mathbf{N} \to \mathbf{B} :: ? \to ? \sqsubseteq f$, because the ascribed function will fail when applied.

13: Recall that :: is a type ascription.

Technically, the adjunction part states that if we have $A \sqsubseteq B$, a term a of type A, and a term b of type B, then $a \sqsubseteq b :: A$ if and only if $a :: B \sqsubseteq b$. The retraction part further states that a is not only more precise than $a :: B :: A$ – which is given by the unit of the adjunction – but is *equi-precise* to it – noted $t \sqsupseteq\!\sqsubseteq t :: B :: A$. Because the DGG dictates that precision implies observational error-approximation, equi-precision implies observational equivalence, and so losing and recovering precision must produce a term that is observationally equivalent to the original one.

[Sie+15]: Siek et al. (2015), *Refined Criteria for Gradual Typing*

[NA18]: New et al. (2018), *Graduality from Embedding-Projection Pairs*

These two approaches to characterizing gradual typing highlight the need to distinguish *syntactic* from *semantic* notions of precision. Indeed, with the usual syntactic precision from Siek et al. [Sie+15], one cannot derive the ep-pair property, in particular the equi-precision stated above. This is why New and Ahmed [NA18] introduce a semantic precision, defined on well-typed terms. This semantic precision serves as a proxy between syntactic precision and the desired observational error-approximation. However, a type-based semantic precision cannot be used for the SGG. Indeed, this theorem[14] requires a notion of precision that *predates* typing: well-typedness of the less precise term is the *consequence* of the theorem. Therefore, a full study of a gradual language that covers SGG, DGG, and embedding-projection pairs needs to consider both syntactic and semantic notions of precision.

14: Not addressed by New and Ahmed [NA18].

Note also that the embedding-projection property does not *per se* imply the DGG: one could pick precision to be the universal relation, which trivially induces ep-pairs, but does not imply observational error-approximation. Conversely, it appears that, in the simply-typed setting considered in prior work, the DGG implies the embedding-projection property. In fact, New and Ahmed [NA18] essentially advocate ep-pairs as an elegant and compositional proof technique to establish the DGG. But as we uncover later on, it turns out that in certain settings – and in particular dependent types – the embedding-projection property imposes *more* desirable constraints on the behaviour of casts than the DGG alone.

In regard of these two remarks, in what follows we use the term *graduality* for the DGG established with respect to a notion of precision which also induces embedding-projection pairs.

9.4. Graduality and Dependent Types

Extending the gradual approach to a setting with full dependent types requires reconsidering several aspects.

9.4.1. Newcomers: the unknown term and the error type

In the simply-typed setting, there is a clear stratification: ? is at the type level, err is at the term level. Likewise, type precision, with ? as greatest element, is distinct from term precision, with err as least element. In the absence of a type/term syntactic distinction as in CIC, this stratification cannot be kept.

Because types permeate terms, ? is no longer only the unknown *type*, but it also acts as an "unknown term". In particular, this makes it possible to consider unknown indices for types, as in Section 9.0.1. More precisely, there is a family of unknown terms $?_A$, indexed by their type A. The traditional unknown type is just $?_\square$, the unknown of the universe $\square$.

Dually, because terms permeate types, we also have the "error type", $\mathrm{err}_\square$. We have to deal with errors in types.

Finally, precision must be unified as a single pre-order, with ? at the top and err at the bottom. The most imprecise term of all[15] is $?_{?_\square}$ – ? for short. At the bottom, err_A is the most precise term of type A.

15: More exactly, there is one such term per universe.

9.4.2. Revisiting safety

The notion of canonical forms used for safety needs to be extended not only with errors as in the simply-typed setting, but also with unknown terms. Indeed, as there is an unknown term $?_A$ inhabiting any type A, we have one new canonical form for each type A. In particular, $?_\mathbf{B}$ cannot possibly reduce to either tt, ff, or $\mathrm{err}_\mathbf{B}$, because doing so would collapse the precision order. Therefore, $?_\mathbf{B}$ should propagate computationally, exactly like $\mathrm{raise}_\mathbf{B}$ in Section 9.2.2 and $\mathrm{err}_\mathbf{B}$.

The difference between errors and unknown terms is not on their dynamic behaviour, but rather on their static interpretation. In essence, the unknown term $?_A$ is a dual form of exceptions: it propagates, but is optimistically comparable – *i.e.* consistent with – any other term of type A. Conversely, err_A should not be consistent with any term of type A. Going back to the issues we identified with the axiomatic (Section 9.2.1) and exceptional (Section 9.2.2) approaches when dealing with type-level comparison, the term

```
head ℕ (? ℕ) (filter ℕ 4 even [ 0 ; 1 ; 2 ; 3 ])
```

is now well-typed: since `S (? ℕ)` is consistent with `? ℕ`, `vec A (? ℕ)` can be deemed consistent with `vec A (S (? ℕ))`. This newly-brought flexibility is the key to support the different scenarios from the introduction. So let us now turn to the question of how to integrate consistency in a dependently-typed setting.

9.4.3. Relaxing conversion

In the simply-typed setting, consistency is a relaxing of syntactic type equality to account for imprecision. In a dependent type theory, there is a more powerful notion than syntactic equality to compare types, namely conversion. The proper notion to relax in the gradual dependently-typed setting is therefore conversion, not syntactic equality.

[GCT16]: Garcia et al. (2016), *Abstracting Gradual Typing*

Garcia, Clark, and Tanter [GCT16] give a general framework for gradual typing that explains how to relax any type predicate to account for imprecision: for a binary type predicate P, its *consistent lifting* $\tilde{P}(A, B)$ holds if there exist static types A' and B' in the denotation[16] of A and B, respectively, such that $P(A', B')$. As observed by [Cas+19], when applied to equality, this defines consistency as a unification problem. Therefore, the consistent lifting of conversion ought to be that two terms t and u are consistently convertible if they denote some static terms t' and u' such that $t' \cong u'$. This is essentially higher-order unification, which is an undecidable problem.

16: Concretization, in abstract interpretation parlance.

[Cas+19]: Castagna et al. (2019), *Gradual Typing: A New Perspective*

It is therefore necessary to adopt some approximation of this relation in order to be able to implement a gradual dependent type theory. There lies an important challenge: because of the dependency of typing on conversion, the static gradual guarantee already demands monotonicity of the approximation one chooses. But if this approximation is defined using reduction, this demand is very close to that of the dynamic gradual guarantee.[17] In practice, this means that the SGG essentially depends on the DGG!

17: In a dependently-typed programming language with separate typing and execution phases, this demand is called the normalization gradual guarantee [ETG19].

[ETG19]: Eremondi et al. (2019), *Approximate Normalization for Gradual Dependent Types*

9.4.4. Dealing with neutrals

Previous work on gradual typing usually only considers reduction on closed terms in order to establish results about the dynamic semantic, such as the DGG. But in dependent type theory, conversion must operate on open terms, and in particular neutral terms such as $1 :: X :: \mathbf{N}$, where X is a type variable, or $x + 1$ where x is of type $\mathbf{N}$ or $?_{\square}$. Such neutral terms cannot reduce further, and can occur in both terms and types. Depending on the upcoming substitution, neutrals can fail, or not. For instance, in $1 :: X :: \mathbf{N}$, if $?_{\square}$ is substituted for X, the term should reduce to 1, but it should fail if $\mathbf{B}$ is substituted instead.

Importantly, less precise variants of neutrals can reduce *more*. For instance, $1 :: ?_{\square} :: \mathbf{N}$ and $?_{\mathbf{N}} + 1$ are respectively less precise than the neutrals above, but do evaluate further – respectively to 1 and to $?_{\mathbf{N}}$. This interaction between neutrals, reduction, and precision spices up the goal of establishing DGG and graduality. In particular, this re-enforces the need to consider a semantic notion of precision, because a too syntactic one is likely not to be stable by reduction: $1 :: X :: \mathbf{N} \sqsubseteq 1 :: ? :: \mathbf{N}$ is obvious syntactically, but $1 :: X :: \mathbf{N} \sqsubseteq 1$ is not.

9.4.5. Dynamic Gradual Guarantee *vs* graduality

In a dependently-typed setting, it is possible to satisfy the DGG while not satisfying the embedding-projection pairs requirement of graduality.

To see why, consider a system in which any term of type A that is not fully-precise immediately reduces to $?_A$. This system would satisfy conservativity, safety, normalization... and the DGG. Indeed, recall that the DGG only requires reduction to be monotone with respect to precision, so using the most imprecise term ? as a universal reduct is surely valid. This collapse of the DGG is impossible in the simply-typed setting because there is no unknown term: it is only possible when $?_A$ exists *as a term*. It is therefore possible to satisfy the DGG while being useless when *computing* with imprecise terms.

On the contrary, the degenerate system breaks the embedding-projection requirement of graduality stated by New and Ahmed [NA18]. For instance, $1 :: ?_{\square} :: \mathbf{N}$ would be convertible to $?_{\mathbf{N}}$, which is *not* observationally equivalent to 1. Therefore, the embedding-projection requirement of graduality goes beyond the DGG in a way that is critical in a dependent type theory, where it captures both the smoothness of the static-to-dynamic checking spectrum, and the proper computational content of valid uses of imprecision.

[NA18]: New et al. (2018), *Graduality from Embedding-Projection Pairs*

9.4.6. Observational refinement

Let us come back to the notion of observational error-approximation used in the simply-typed setting to state the DGG. New and Ahmed [NA18] justify this notion because in "gradual typing we are not particularly interested in when one program diverges more than another, but rather when it produces more type errors".

This point of view is adequate in the simply-typed setting because the addition of ascriptions may only produce more type errors; in particular, adding ascriptions can never lead to divergence when the original term does not diverge itself. Thus, in that setting, the definition of observational error-approximation includes equi-divergence.

The situation in the dependent setting is however more complicated if the theory admits divergence.[18] In a gradual dependent type theory that admits divergence, a diverging term is more precise than the unknown term ?. Because the unknown term does not diverge, this breaks the left-to-right implication of equi-divergence. Note that this argument does not rely on any specific definition of precision, just on the fact that the unknown is a term, and not just a type.

Additionally, an error at a diverging type X may be ascribed to $?_{\square}$, then back to X. Evaluating this roundtrip requires evaluating X itself, which makes the less precise term diverge. This breaks the right-to-left implication of equi-divergence.

To summarize, the way to understand these counterexamples is that in a dependent and non-terminating setting, the motto of graduality ought to be adjusted: more precise programs produce more type errors *or diverge more*. This leads to the following definition of observational refinement.

18: There exist non-gradual dependently-typed programming languages that admit divergence, *e.g.* Dependent Haskell [Eis16] or Idris [Bra13]. We will also present one such theory in this article.

[Eis16]: Eisenberg (2016), *Dependent Types in Haskell: Theory and Practice*

[Bra13]: Brady (2013), *Idris, a general-purpose dependently typed programming language: Design and implementation*

Definition 9.4. *Observational refinement*

> A term $\Gamma \vdash t : A$ observationally refines a term $\Gamma \vdash u : A$, noted $t \sqsubseteq^{\text{ob}} u$, if for all boolean-valued observation context $\mathcal{C} :(\Gamma \vdash A) \Rightarrow (\vdash \mathbf{B})$

> closing over all free variables, if $\mathcal{C}[u] \to^\star \mathrm{err}_\mathbf{B}$ or diverges, then either $\mathcal{C}[t] \to^\star \mathrm{err}_\mathbf{B}$ or $\mathcal{C}[t]$ diverges.

The main difference with observational error-approximation is that in this definition, errors and divergence are collapsed. In particular, equi-refinement does *not* imply observational equivalence, because one term might diverge while the other reduces to an error. Happily, if the gradual dependent theory is strongly normalizing, both notions observational error-approximation $\preccurlyeq^{\mathrm{ob}}$ and observational refinement $\sqsubseteq^{\mathrm{ob}}$ coincide.

9.5. The Fire Triangle of Graduality

To sum up, we have so far seen four important properties that can be expected from a gradual type theory: safety, conservativity with respect to a given static system, graduality, and normalization. Any type theory ought to satisfy at least safety. Unfortunately, we now show that mixing the three other properties is impossible for STLC, and *a fortiori* for CIC.

9.5.1. Preliminary: regular reduction

To derive this general impossibility result by relying only on the properties and without committing to a specific language or theory, we need to assume that the reduction system used to decide conversion is "regular". This means that it only looks at the weak-head normal forms of sub-terms for reduction rules, and does not magically shortcut reduction, for instance based on the specific syntax of inner terms. As an example, β-reduction is not allowed to look into the body of the lambda term to decide how to proceed.

This property is satisfied in all actual systems we know of, but formally stating it in full generality, in particular without devoting to a particular syntax, is beyond our current scope. Fortunately, in the following, we rely only on a much weaker hypothesis, which is a slight strengthening of the retraction hypothesis of embedding-projection pairs. Recall that retraction says that when $A \sqsubseteq B$, any term t of type A is equi-precise to $t :: B :: A$.

We additionally require that for any context $\mathcal{C}$, if $\mathcal{C}[t]$ reduces at least k steps, then $\mathcal{C}[t :: B :: A]$ also reduces at least k steps. Intuitively, this means that the reduction of $\mathcal{C}[t :: B :: A]$, while free to decide when to get rid of the embedding-to-B-projection-to-A, cannot use it to avoid reducing t. This property is true in all gradual languages, where type information at runtime is used only as a monitor.

9.5.2. Gradualizing STLC

Let us first consider the case of STLC. We show that Ω is *necessarily* a well-typed, diverging term in any gradualization of STLC that satisfies the other properties.

Theorem 9.5. *Fire Triangle of Graduality* for STLC

> Suppose a gradual type theory that satisfies both conservativity with respect to STLC and graduality. Then it cannot be normalizing.

Proof.

We pose $\Omega \coloneqq \delta\ (\delta :: ?)$ with $\delta \coloneqq \lambda\, x{:}?.\ (x :: ? \to ?)\ x$ and show that it must necessarily be a well-typed, diverging term. Because the unknown type $?$ is consistent with any type (Section 9.3) and $? \to ?$ is a valid type (by conservativity), the self-applications in Ω are well-typed, δ has type $? \to ?$, and Ω has type $?$. Now, we remark that $\Omega = \mathcal{C}[\delta]$ with $\mathcal{C}[\cdot] \coloneqq [\cdot]\ (\delta :: ?)$.

We show by induction on k that Ω reduces at least k steps, the initial case being trivial. Suppose that Ω reduces at least k steps. By maximality of $?$ with respect to precision, we have that $? \to ? \sqsubseteq ?$, so we can apply the strengthening of graduality applied to δ, which tells us that $\mathcal{C}[\delta :: ? :: ? \to ?]$ reduces at least k steps, because $\mathcal{C}[\delta]$ reduces at least k steps.

But Ω reduces in one step of β-reduction to $\mathcal{C}[\delta :: ? :: ? \to ?]$. So Ω reduces at least $k + 1$ steps.

This means that Ω diverges, which is a violation of normalization. □

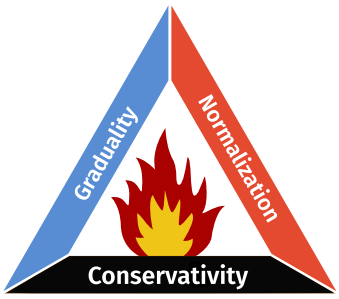

Figure 9.1. The Fire Triangle of Graduality

This result could be extended to all terms of the untyped lambda calculus, not only Ω, in order to obtain the embedding theorem of GTLC [Sie+15]. Therefore, the embedding theorem is not an independent property, but rather a consequence of conservativity and graduality. This is why we have not included it in our overview of the gradual approach in Section 9.3.

[Sie+15]: Siek et al. (2015), *Refined Criteria for Gradual Typing*

9.5.3. Gradualizing CIC

We can now prove the same impossibility theorem for CIC, by reducing it to the case of STLC. In general, this theorem can be proven for type theories others than CIC, as soon as they faithfully embed STLC.

Theorem 9.6. *Fire Triangle of Graduality* for CIC

> A gradual dependent type theory cannot simultaneously satisfy conservativity with respect to CIC, graduality and normalization.

Proof.

We show that a gradual dependent type theory satisfying CIC and graduality must contain a diverging term, thus contravening normalization. The typing rules of CIC contain the typing rules of STLC, using only one universe $\square_0$, and the notions of reduction coincide, so CIC embeds STLC. This is a well-known result on Pure Type Systems [Bar91], of which CC_ω is one of many examples. This means that conservativity with respect to CIC implies conservativity with respect to STLC.

[Bar91]: Barendregt (1991), *An Introduction to Generalized Type Systems*

Additionally, graduality can be specialized to the simply-typed fragment of the theory, by setting the unknown type $?$ to be $?_{\square_0}$. We can then

apply Theorem 9.5, and get a diverging well-typed term, finishing the proof. □

9.5.4. The Fire Triangle in practice

In non-dependent settings, all gradual languages where ? is universal admit non-termination and therefore compromise normalization. Garcia and Tanter [GT20] discuss the possibility to gradualize STLC without admitting non-termination, for instance by considering that ? is not universal and denotes only base types[19]. Without sacrificing the universal unknown type, one could design a variant of GTLC that uses some mechanism to detect divergence, such as termination contracts Nguyễn et al. [Ngu+19]. This would yield a language that certainly satisfies normalization, but it would break graduality. Indeed, because the contract system is necessarily under-approximating in order to be sound – and actually imply normalization –, there are effectively-terminating programs with imprecise variants that yield termination contract errors.

[GT20]: Garcia et al. (2020), *Gradual Typing as if Types Mattered*

19: In such a case, $? \to ? \not\sqsubseteq ?$, so our argument involving Ω is invalid.

[Ngu+19]: Nguyễn et al. (2019), *Size-Change Termination as a Contract: Dynamically and Statically Enforcing Termination for Higher-Order Programs*

To date, the only related work that considers the gradualization of full dependent types with ? as both a term and a type, is the work on *GDTL* [ETG19]. GDTL is a programming language with a clear separation between the typing and execution phases, like Idris [Bra13]. GDTL adopts a different strategy in each phase: for typing, it uses *Approximate Normalization*, which always produces $?_A$ as a result of going through imprecision and back. This implies that the system is normalizing – and thus that conversion is decidable –, but it breaks graduality for the same reason as the degenerate system we discussed in Section 9.4[20]. In such a phased setting, the lack of computational content of Approximate Normalization is not critical, because it only means that typing becomes overly optimistic. To execute programs, GDTL relies on standard GTLC-like reduction semantics, which is computationally precise, but not normalizing.

[ETG19]: Eremondi et al. (2019), *Approximate Normalization for Gradual Dependent Types*

[Bra13]: Brady (2013), *Idris, a general-purpose dependently typed programming language: Design and implementation*

20: The example uses a gain of precision from the unknown type to $\mathbb{N}$, so it behaves just the same in GDTL

9.6. GCIC: An Overview

Given the Fire Triangle of Graduality (Theorem 9.6), we know that gradualizing CIC implies making some compromise. Instead of focusing on one possible solution, we actually develop a common parametrized framework, GCIC, where the parameters control which of the three properties – normalization, graduality and conservativity – is compromised. This section gives an informal, non-technical overview of this system, highlighting the main challenges and results.

9.6.1. Three in one

Two parameters... To explore the spectrum of possibilities opened by the Fire Triangle of Graduality, we develop a general approach to gradualizing CIC, and use it to define three theories, corresponding to different resolutions of the triangular tension between normalization, graduality and conservativity with respect to CIC.

The crux of our approach is to recognize that, while there is not much to vary within STLC itself to address the tension of the Fire Triangle of

Graduality, there are several variants of CIC that can be considered by changing the hierarchy of universes and its impact on typing – after all, its core CC_ω is but a particular Pure Type System [Bar91]. Thus, we consider a parametrized version of a gradual CIC, called GCIC, with two parameters[21].

[Bar91]: Barendregt (1991), *An Introduction to Generalized Type Systems*

21: This system is precisely detailed in Figure 10.2

The first parameter characterizes how the universe level of a Π-type is determined in typing rules: either as taking the *maximum* of the levels of the involved types – as in standard CIC – or as the *successor* of that maximum. The latter option yields a variant of CIC that we call $CIC^{\uparrow}$ – read "CIC-shift". $\text{CIC}^{\uparrow}$ is a subset of CIC, with a stricter constraint on universe levels. In particular $\text{CIC}^{\uparrow}$ loses the closure of universes under dependent functions that CIC enjoys. As a consequence, some well-typed CIC terms are not well-typed in $\text{CIC}^{\uparrow}$.[22]

22: A typical example of a well-typed CIC term that is ill typed in $\text{CIC}^{\uparrow}$ is $\text{nArrow} : \mathbb{N} \to \square$, where $\text{nArrow}\, n$ is the type of functions that accept n arguments. Such dependent arities violate the universe constraint of $\text{CIC}^{\uparrow}$.

The second parameter is the dynamic counterpart of the first parameter: its role is to control universe levels during the reduction of type casts between Π-types. We only allow this reduction parameter to be loose – *i.e.* using maximum – if the typing parameter is also loose. Indeed, letting the typing parameter be strict – *i.e.* using successor of the maximum – while the reduction parameter is loose breaks subject reduction, and hence safety.

... and three meaningful theories. Based on these parameters, we develop the following three variants of GCIC, whose properties are summarized in Figure 9.2 – because GCIC is one common parametrized framework, we are able to establish most properties for all variants at once.

The first variant, $GCIC^{\mathcal{G}}$, is a theory that satisfies both conservativity with respect to CIC and graduality, but sacrifices normalization. This theory is a rather direct application of the principles discussed in Section 9.4 by extending CIC with errors and unknown terms, and replacing conversion with consistency. This results in a theory that is not normalizing.

Next, $GCIC^{\uparrow}$ satisfies both normalization and graduality, and supports conservativity, but only with respect to $\text{CIC}^{\uparrow}$. This theory uses the universe hierarchy at the *typing level* to detect and forbid the potential non-termination induced by the use of consistency instead of conversion.

Finally, $GCIC^{\mathcal{N}}$ satisfies both conservativity with respect to CIC and normalization, but does not fully validate graduality. This theory uses the universe hierarchy at the *computational level* to detect potential divergence, eagerly raising errors. Such runtime failures invalidate the DGG for some terms, and hence graduality, as well as the SGG, since in our dependent setting it depends on the DGG.

	Safety	Normalization	Conservativity wrt.	Graduality	SGG	DGG
$\text{GCIC}^{\mathcal{G}}$	✓	✗	CIC	✓	✓	✓
$\text{GCIC}^{\uparrow}$	✓	✓	$\text{CIC}^{\uparrow}$	✓	✓	✓
$\text{GCIC}^{\mathcal{N}}$	✓	✓	CIC	✗	✗	✗

Figure 9.2. GCIC variants and their properties

Practical implications of GCIC variants. Regarding our introductory examples, all three variants of GCIC support the exploration of the type-level precision spectrum. In particular, we can define `filter` by giving it the imprecise type

```
forall A n (p : A → 𝔹), vec A n → vec A (? ℕ)
```

in order to bypass the difficulty of precisely characterizing the size of the output vector. Any invalid optimistic assumption is detected during reduction and reported as an error.

Unsurprisingly, the semantic differences between the three GCIC variants crisply manifest in the treatment of potential non-termination, more specifically, *self application*. Let us come back to the term Ω used in the proof of Theorem 9.6. In all three variants, this term is well-typed. In $\text{GCIC}^{\mathcal{G}}$, it reduces forever, as it would in the untyped lambda calculus: $\text{GCIC}^{\mathcal{G}}$ can embed the untyped lambda calculus, just as GTLC [Sie+15]. In $\text{GCIC}^{\mathcal{N}}$, this term fails at runtime because of the strict universe check in the reduction of casts, which breaks graduality because $?_{\square_i} \to ?_{\square_i} \sqsubseteq ?_{\square_i}$ tells us that the upcast-downcast coming from an ep-pair should not fail. In $\text{GCIC}^{\uparrow}$, Ω fails in the same way as in $\text{GCIC}^{\mathcal{N}}$, but this does not break graduality because of the shifted universe level on Π-types. Indeed, a consequence of this stricter typing rule is that in $\text{GCIC}^{\uparrow}$, $?_{\square_i} \to ?_{\square_i} \sqsubseteq ?_{\square_j}$ for any $j > i$, but $?_{\square_i} \to ?_{\square_i} \not\sqsubseteq ?_{\square_j}$. Therefore, the casts performed in Ω do not come from an ep-pair any more, and can thus legitimately fail. This is described in full details in Section 10.2.3.

[Sie+15]: Siek et al. (2015), *Refined Criteria for Gradual Typing*

Another scenario where the differences in semantics manifest is functions with *dependent arities*. For instance, the well-known C function `printf` can be embedded in a well-typed fashion in CIC: it takes as first argument a format string and computes from it both the type and *number* of later arguments. In $\text{GCIC}^{\mathcal{G}}$ it can be gradualized as much as one wants, without surprises. This function, however, brings into light the limitation of $\text{GCIC}^{\uparrow}$: since the format string can specify an arbitrary number of arguments, we need as many $\to$, and `printf` cannot be well-typed in a theory where universes are not closed under function types. In $\text{GCIC}^{\mathcal{N}}$, `printf` is well-typed, but the same problem will appear dynamically when casting `printf` to $?$ and back to its original type: the result will be a function that works only on format strings specifying no more arguments than the universe level at which it has been typed. Note that this constitutes an example of violation of graduality for $\text{GCIC}^{\mathcal{N}}$, even of the dynamic gradual guarantee.

Which variant to pick? As explained in the introduction, the aim here is to shed light on the design space of gradual dependent type theories, not to advocate for one specific design. The appropriate choice indeed depends on the specific goals of the language designer, or perhaps more pertinently, on the specific goals of a given project, at a specific point in time. The key characteristics of each variant are as follows.

[Eis16]: Eisenberg (2016), *Dependent Types in Haskell: Theory and Practice*

$\text{GCIC}^{\mathcal{G}}$ favours flexibility over decidability of type-checking. While this might appear heretical in the context of proof assistants, this choice has been embraced by practical languages such as Dependent Haskell [Eis16], where both divergence and runtime errors can happen at the type level.

The pragmatic argument is simplicity: by letting programmers be responsible, there is no need for termination checking techniques and other restrictions.

$\text{GCIC}^{\uparrow}$ is theoretically pleasing as it enjoys both normalization and graduality. In practice, though, the fact that it is not conservative with respect to full CIC means that one would not be able to simply import existing libraries as soon as they fall outside the $\text{CIC}^{\uparrow}$ subset. In $\text{GCIC}^{\uparrow}$, the introduction of ? should be done with an appropriate understanding of universe levels. This might not be a problem for advanced programmers, but would surely be harder to grasp for beginners.

Finally, $\text{GCIC}^{\mathcal{N}}$ is normalizing and able to import existing libraries without restrictions, at the expense of some surprises on the graduality front. Programmers would have to be willing to accept that they cannot just sprinkle ? as they see fit without further consideration, as any dangerous usage of imprecision will be flagged during conversion.

In the same way that systems like Coq, Agda or Idris support different ways to customize their semantics regarding termination,[23] and of course, many programming languages implementations supporting some sort of customization[24] one can imagine a flexible realization of GCIC that give users the control over the two parameters we identify in this work, and therefore lets them access all three GCIC variants. Considering the inherent tension captured by the Fire Triangle of Graduality, such a pragmatic approach might be the most judicious choice, making it possible to gather experience and empirical evidence about the pros and cons of each in a variety of concrete scenarios.

23: With the possibility to allow $\square : \square$, switch off termination checking, use the partial/total compiler flags...

24: GHC is a salient representative.

9.6.2. Typing, conversion and bidirectional elaboration

As explained in Section 9.3, in a gradual language, whenever we reclaim precision, we might be wrong and need to fail in order to preserve safety. In a simply-typed setting, the standard approach is to define typing on a gradual source language, and then to translate terms via a type-directed elaboration to a target *cast calculus*, *i.e.* a language with explicit runtime type checks. This elaboration inserts casts, needed for a well-behaved reduction [ST06]. For instance, in a call-by-value language, the upcast (loss of precision) $\langle ? \Leftarrow \mathbf{N} \rangle\, 10$ is considered a (tagged) value, and the downcast (gain of precision) $\langle \mathbf{N} \Leftarrow ? \rangle\, v$ reduces successfully if v is such a tagged natural number, or to an error otherwise.

[ST06]: Siek et al. (2006), *Gradual Typing for Functional Languages*

We follow a similar approach for GCIC, which is elaborated in a type-directed manner to a second calculus, named *CastCIC* (Section 10.1). The interplay between typing and cast insertion is however more subtle in the context of a dependent type theory. Because typing needs computation, and reduction is only meaningful in the target language, CastCIC is used *as part of the elaboration* in order to compare types (Section 10.2). This means that GCIC has no typing on its own, independent of its elaboration to CastCIC.[25]

25: This is similar to what happens in practice in proof assistants such as Coq [Coq22b, Core language], where terms input by the user in the Gallina language are first elaborated in order to add implicit arguments, coercions, etc. The computation steps required by conversion are performed on the elaborated terms, never on the raw input syntax.

[Coq22b]: Coq Development Team (2022), *The Coq proof assistant reference manual*

In order to satisfy conservativity with respect to CIC, ascriptions in GCIC are required to satisfy consistency. For instance, $\mathrm{tt} :: ? :: \mathbf{N}$ is well-typed by consistency – used twice –, but $\mathrm{tt} :: \mathbf{N}$ is ill-typed. Such ascriptions in CastCIC are realized by casts. For instance $0 :: ? :: \mathbf{B}$ in GCIC elaborates – up to desugaring and reduction – to $\langle \mathbf{B} \Leftarrow ?_{\square} \rangle\, \langle ?_{\square} \Leftarrow \mathbf{N} \rangle\, 0$ in CastCIC. A major

difference between ascriptions in GCIC and casts in CastCIC is that casts are not required to satisfy consistency: a cast between any two types is well-typed, although of course it might produce an error.

This is where the bidirectional structure is crucial. First, it is required in order to tame the non-transitive consistency relation. Indeed, in the previous example of $\mathrm{tt} :: \mathbf{N}$, if one kept a free-standing rule like Rule Conv and simply replaced conversion by consistency, one could use the rule twice, through ?, and the term would be well-typed. But consistency demands that only terms with explicitly-ascribed imprecision enjoy its flexibility. This observation is standard in the gradual typing literature [ST06; ST07; GCT16], but becomes even more crucial in the context of gradual dependent types [ETG19]. Moreover, the bidirectional structure is very suited to the description of a type-based elaboration, and directly translates to a deterministic typing/elaboration algorithm for GCIC.

[ST06]: Siek et al. (2006), *Gradual Typing for Functional Languages*
[ST07]: Siek et al. (2007), *Gradual Typing for Objects*
[GCT16]: Garcia et al. (2016), *Abstracting Gradual Typing*
[ETG19]: Eremondi et al. (2019), *Approximate Normalization for Gradual Dependent Types*

9.6.3. Precisions and properties

As explained earlier (Section 9.4), we need three different notions of precision to deal with SGG and graduality.

At the source level – GCIC –, we introduce a notion of *syntactic precision*, that captures the intuition of a more imprecise term as "the same term with sub-terms and/or type annotations replaced by ?", and is defined without any assumption of typing. In CastCIC, we define a notion of *structural precision*, which is mostly syntactic except that, in order to account for cast insertion during elaboration, it tolerates precision-preserving casts. For instance, $\langle A \Leftarrow A \rangle\, t$ is related to t by structural precision.

Armed with these two notions of precision, we prove elaboration graduality (Theorem 10.23), which is the equivalent of the static gradual guarantee in our setting: if a term t of GCIC elaborates to a term t' of CastCIC, then a term u less syntactically precise than t in GCIC elaborates to a term u' less structurally precise than t' in CastCIC. Because DGG is about the behaviour of terms during reduction, it is technically stated and proven for CastCIC. We show in Section 10.4 that DGG can be proven for CastCIC – in its variants $\text{CastCIC}^{\mathcal{G}}$ and $\text{CastCIC}^{\uparrow}$ – on structural precision.

However, as explained in Section 9.3, we cannot expect to prove graduality for these CastCIC variants with respect to structural precision directly. In order to overcome this problem, and to justify the design of CastCIC, we build two kinds of models for CastCIC. The first[26] is a syntactic model [Bou18] – akin to a program translation or a compilation phase –, and is used to justify the reduction rules and prove that they are terminating. The second[27] endows types with the structure of an ordered set, or poset. This makes it possible to reason about the semantic notion of *propositional precision* and prove that it gives rise to embedding-projection pairs, thereby establishing graduality. These models are described in Section 11.1.

26: That we call the *discrete model*.
[Bou18]: Boulier (2018), *Extending Type Theory with Syntactical Models*
27: That we call the *monotone model*

From GCIC to CastCIC: Bidirectional Elaboration

10.

10.1 CastCIC 129
10.1.1 System definition 129
10.1.2 Safety 133
10.2 Bidirectional Elaboration 135
10.2.1 System definition 135
10.2.2 Direct properties 138
10.2.3 Illustration: back to Ω 140
10.3 Simulation 141
10.3.1 Structural precision 142
10.3.2 Catch-up lemmas 144
10.3.3 Simulation 146
10.4 Properties of GCIC 147
10.4.1 Conservativity 148
10.4.2 Elaboration Graduality 149
10.4.3 Dynamic Gradual Guarantee 150

Let us now look in details at the elaboration from the source gradual system GCIC to the target cast calculus CastCIC. We start with CastCIC, describing its typing, reduction and metatheoretical properties (Section 10.1). Next, we describe GCIC and its bidirectional elaboration to CastCIC, along with a few direct properties (Section 10.2). This elaboration can be seen as an extension of the bidirectional presentation of CIC. To illustrate the semantics of the different GCIC variants, we show how the Ω term (Section 10.2.3) behaves in them. We finally expose technical properties of the reduction of CastCIC (Section 10.3) used to prove the most important theorems on elaboration: conservativity over CIC or $\text{CIC}^{\uparrow}$, as well as the gradual guarantees (Section 10.4).

In this whole chapter, we do not treat indexed inductive types, thus the system should be seen as an extension of CIC$_-$, rather than full-blown CIC. We come back to this issue in Chapter 11. The original reference [Len+22] considers the case of general inductive types, here we restrict the presentation to **Li** to ease readability.

[Len+22]: Lennon-Bertrand et al. (2022), *Gradualizing the Calculus of Inductive Constructions*

10.1. CastCIC

10.1.1. System definition

Syntax. The syntax of *CastCIC*[1] extends that of CIC$_-$ with three new term constructors: the *unknown term* $?_T$ and *dynamic error* err_T of type T, as well as the *cast* $\langle T \Leftarrow S \rangle\, t$ of a term t of type S to type T

$$\mathrm{Term}_{\text{CastCIC}} \ni t := \cdots \mid ?_t \mid \mathrm{err}_t \mid \langle t \Leftarrow t \rangle\, t \qquad \text{(Syntax of CastCIC)}$$

with casts associating to the right: $\langle S' \Leftarrow S \rangle \langle T \Leftarrow T' \rangle\, t$ corresponds to the fully-parenthesized $\langle S' \Leftarrow S \rangle\, (\langle T \Leftarrow T' \rangle\, t)$. We also collapse successive ones: $\langle T'' \Leftarrow T' \Leftarrow T \rangle\, t$ is shorthand for $\langle T'' \Leftarrow T' \rangle \langle T' \Leftarrow T \rangle\, t$. The unknown term and dynamic error both behave as exceptions as defined in ExTT [PT18]. Casts keep track of the use of consistency during elaboration, implementing a form of runtime type-checking, raising the error err_T in case of a type mismatch. We call *static* the terms of CastCIC that do not use any of these new constructors – static CastCIC terms thus correspond to CIC terms.

1: Written using a dark blue colour.

[PT18]: Pédrot et al. (2018), *Failure is Not an Option An Exceptional Type Theory*

Universe parameters. CastCIC is parametrized by two functions, described in Figure 10.1, to account for the three different variants of GCIC we consider – see Section 9.6.1 : $\text{CastCIC}^{\mathcal{G}}$, $\text{CastCIC}^{\uparrow}$ and $\text{CastCIC}^{\mathcal{N}}$. The

$$\begin{array}{lll}
\mathrm{s}_\Pi(i,j) := \max(i,j) & \mathrm{c}_\Pi(i) := i & (\text{GCIC}^{\mathcal{G}}\text{-CastCIC}^{\mathcal{G}}) \\
\mathrm{s}_\Pi(i,j) := \max(i,j) & \mathrm{c}_\Pi(i) := i-1 & (\text{GCIC}^{\mathcal{N}}\text{-CastCIC}^{\mathcal{N}}) \\
\mathrm{s}_\Pi(i,j) := \max(i,j)+1 & \mathrm{c}_\Pi(i) := i-1 & (\text{GCIC}^{\uparrow}\text{-CastCIC}^{\uparrow})
\end{array}$$

Figure 10.1. Universe parameters

$$\textsc{ΠTy}\ \frac{\Gamma \vdash A \rhd_\square \square_i \qquad \Gamma \vdash B \rhd_\square \square_j}{\Gamma \vdash \Pi\, x{:}\, A.\ B \rhd \square_{s_\Pi(i,j)}} \qquad \textsc{Unk}\ \frac{\Gamma \vdash T \rhd_\square \square_i}{\Gamma \vdash ?_T \rhd T}$$

$$\textsc{Err}\ \frac{\Gamma \vdash T \rhd_\square \square_i}{\Gamma \vdash \mathrm{err}_T \rhd T} \qquad \textsc{Cast}\ \frac{\Gamma \vdash T \rhd_\square \square \qquad \Gamma \vdash T' \rhd_\square \square \qquad \Gamma \vdash t \lhd T}{\Gamma \vdash \langle T' \Leftarrow T \rangle\, t \rhd T'}$$

Figure 10.2. Typing rules for CastCIC (Extending those for CIC, replace Rule ΠTy)

first function s_Π computes the level of the universe of a dependent function type, given the levels of its domain and codomain – see the updated Rule ΠTy in Figure 10.2. The second function c_Π controls the universe level in the reduction of a cast between $? \to ?$ and $?$ – see Figures 10.3 and 10.4d.

Typing. The first difference between CastCIC and CIC is Rule ΠTy, given in Figure 10.2, which uses the s_Π parameter. In CastCIC$^{\mathcal{G}}$ and CastCIC$^{\mathcal{N}}$, this rule corresponds to the usual one of CIC, but in CastCIC$^{\uparrow}$ it is stricter. All other typing rules are exactly the same as in CIC.

Next, Rules Unk and Err say that both $?_T$ and err_T infer T when T is a type.

Finally, Rule Cast ensures that both the source and target of the cast are indeed types, and that the cast term indeed has the source type. Note that in CastCIC, as is sometimes the case in cast calculi [SW10; NA18] no consistency premise is required for a cast to be well-typed. Here, consistency only plays a role in GCIC, but completely disappears after elaboration. Instead, CastCIC relies only on standard (algorithmic) conversion.

[SW10]: Siek et al. (2010), *Threesomes, with and without Blame*
[NA18]: New et al. (2018), *Graduality from Embedding-Projection Pairs*

$$\mathrm{Head} \ni h ::= \square_i \mid \Pi \mid \mathbf{Li}$$

$$\mathrm{head}\,(\Pi\, x{:}\, A.\ B) ::= \Pi \qquad \mathrm{head}\,(\square_i) ::= \square_i \qquad \mathrm{head}\,(\mathbf{Li}\,(A)) ::= \mathbf{Li}$$

$$\mathrm{germ}_i \square_j ::= \begin{cases} \square_j & \text{if } j < i \\ \mathrm{err}_{\square_i} & \text{if } j \geq i \end{cases} \qquad \mathrm{germ}_i\, \mathbf{Li} ::= \mathbf{Li}\,(?_{\square_i})$$

$$\mathrm{germ}_i\, \Pi ::= \begin{cases} ?_{\square_{c_\Pi(i)}} \to ?_{\square_{c_\Pi(i)}} & \text{if } c_\Pi(i) \geq 0 \\ \mathrm{err}_{\square_i} & \text{if } c_\Pi(i) < 0 \end{cases}$$

Figure 10.3. Head constructor and germ

Reduction. The typing rules provide little insight on the new primitives; the interesting part really lie in their reduction behaviour.

Reduction relies on two auxiliary functions relating *head constructors* $h \in \mathrm{Head}$ to those terms that start with either Π, $\square$ or and inductive type – in our running example, $\mathbf{Li}$ – the set of which we call $\mathrm{Type}_{\mathrm{CastCIC}}$. These are defined in Figure 10.3. The first is the function head, which returns the head constructor of a type.

In the other direction, the *germ* function $\mathrm{germ}_i h$ constructs the least precise type with head h at level i. In the case where no such type exists —

e.g. when $c_\Pi(i) < 0$ – this least precise type is the error. The germ function corresponds to an abstraction function in the sense of AGT [GCT16], if one interprets the head h as the set of all types whose head type constructor is h. Wadler and Findler [WF09] christened the corresponding notion a *ground type*, later reused in the gradual typing literature. This terminology however clashes with its prior use in denotational semantics [Lev04]: there a ground type is a first-order datatype. Note also that Siek and Taha [ST06] call ground types the base types of the language, such as $\mathbf{B}$ and $\mathbf{N}$. We therefore prefer the less overloaded term germ, used by analogy with the geometrical notion of the *germ of a section* [MM94]: the germ of a head constructor represents an equivalence class of types that are locally the same.

[GCT16]: Garcia et al. (2016), *Abstracting Gradual Typing*

[WF09]: Wadler et al. (2009), *Well-Typed Programs Can't Be Blamed*

[Lev04]: Levy (2004), *Call-By-Push-Value: A Functional/Imperative Synthesis*

[ST06]: Siek et al. (2006), *Gradual Typing for Functional Languages*

[MM94]: Mac Lane et al. (1994), *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*

The exact design of the reduction rules is mostly dictated by the models of CastCIC presented later in Section 11.1. Nevertheless, we now provide some intuition about their meaning. We only present here the rules for top-level reduction:[2] the congruence closure to obtain full reduction is completely standard. As for weak-head reduction, we give the adequate contextual closure later on when we prove progress.

2: As all our reduction rules have empty premises, we spare the needless bar to make them more readable.

$$\textsc{Π-Unk}: ?_{\Pi(x:A).\,B} \rightharpoonup \lambda\, x\colon A.\, ?_B \qquad \textsc{Π-Err}: \mathrm{err}_{\Pi\, x:A.\, B} \rightharpoonup \lambda\, x\colon A.\, \mathrm{err}_B$$

$$\textsc{Match-Unk}: \mathrm{ind}_{\mathbf{Li}}\,(?_{\mathbf{Li}\,(A)}; z.P; b_\varepsilon, y_1.y_2.p_{y_2}.b_{::}) \rightharpoonup ?_{P[z \coloneqq ?_{\mathbf{Li}\,(A)}]}$$

$$\textsc{Match-Err}: \mathrm{ind}_{\mathbf{Li}}\,(\mathrm{err}_{\mathbf{Li}\,(A)}; z.P; b_\varepsilon, y_1.y_2.p_{y_2}.b_{::}) \rightharpoonup \mathrm{err}_{P[z \coloneqq \mathrm{err}_{\mathbf{Li}\,(A)}]}$$

$$\textsc{List-Unk}: \langle \mathbf{Li}\,(A') \Leftarrow \mathbf{Li}\,(A'')\rangle\, ?_{\mathbf{Li}\,(A)} \rightharpoonup ?_{\mathbf{Li}\,(A')} \qquad \textsc{List-Err}: \langle \mathbf{Li}\,(A') \Leftarrow \mathbf{Li}\,(A'')\rangle\, \mathrm{err}_{\mathbf{Li}\,(A)} \rightharpoonup \mathrm{err}_{\mathbf{Li}\,(A')}$$

$$\textsc{Down-Unk}: \langle X \Leftarrow ?_\square\rangle\, ?_{?_\square} \rightharpoonup ?_X \qquad \textsc{Down-Err}: \langle X \Leftarrow ?_\square\rangle\, \mathrm{err}_{?_\square} \rightharpoonup \mathrm{err}_X$$

Figure 10.4a. Propagation rules for ? and err

The first set of rules, given in Figure 10.4a, specify the exception-like propagation behaviour of both ? and err at function and inductive types. Rules List-Unk and List-Err similarly propagate ? and err when cast between the same inductive type, and Rules Down-Unk and Down-Err do the same from the unknown type to any type X.

$$\Pi\text{-}\Pi: \langle \Pi\, y\colon A_2.\, B_2 \Leftarrow \Pi\, x\colon A_1.\, B_1\rangle(\lambda\, x\colon A.\, t) \rightharpoonup \lambda\, y\colon A_2.\, \langle B_2 \Leftarrow B_1[x \coloneqq \langle A_1 \Leftarrow A_2\rangle\, y]\rangle(t[x \coloneqq \langle A \Leftarrow A_2\rangle\, y])$$

$$\textsc{Univ-Univ}: \langle \square_i \Leftarrow \square_i\rangle\, A \rightharpoonup A \qquad \textsc{Nil-Nil}: \langle \mathbf{Li}\,(A_2) \Leftarrow \mathbf{Li}\,(A_1)\rangle\, \varepsilon_A \rightharpoonup \varepsilon_{A_2}$$

$$\textsc{Cons-Cons}: \langle \mathbf{Li}\,(A_2) \Leftarrow \mathbf{Li}\,(A_1)\rangle\, a ::_A l \rightharpoonup \langle A_2 \Leftarrow A_1\rangle\, a ::_{A_2} \langle \mathbf{Li}\,(A_2) \Leftarrow \mathbf{Li}\,(A_1)\rangle\, l$$

Figure 10.4b. Success rules for casts

Next come the rules of Figure 10.4b, which correspond to success cases of dynamic checks, where the cast is between types with the same head. In that case, casts are either completely erased when possible, or propagated. As usual in gradual typing, directly inspired by higher-order contracts [FF02], Rule Π-Π distributes the function cast in two casts, one for the argument and one for the body; note the substitution in the source codomain in order to account for dependency. Also, because constructors

[FF02]: Findler et al. (2002), *Contracts for Higher-Order Functions*

and inductive types are fully applied, this Π-Π rule cannot be blocked because of a partially-applied constructor or inductive. Regarding inductive types, the propagation of casts on sub-terms cannot be avoided in the list type, but if we follow this strategy for simpler inductive types, *e.g.* $\mathbb{N}$, the restriction to reduce only on constructors means that a cast between $\mathbb{N}$ and $\mathbb{N}$ is blocked until its argument term is a constructor, rather than disappearing right away as for $\Box$. This is somewhat non-optimal, but we stick to it here for simplicity.

$$\textsc{Head-Err}\ \frac{T, T' \in \mathrm{Type}_{\mathrm{CastCIC}} \qquad \mathrm{head}\, T \neq \mathrm{head}\, T'}{\langle T' \Leftarrow T \rangle\, t \rightharpoonup \mathrm{err}_{T'}}$$

$$\textsc{Dom-Err}:\ \langle T \Leftarrow \mathrm{err}_{\Box} \rangle\, t \rightharpoonup \mathrm{err}_{T} \qquad\qquad \textsc{Codom-Err}\ \frac{T \in \mathrm{Type}_{\mathrm{CastCIC}}}{\langle \mathrm{err}_{\Box} \Leftarrow T \rangle\, t \rightharpoonup \mathrm{err}_{\mathrm{err}_{\Box}}}$$

Figure 10.4c. Failure rules for casts

On the contrary, Figure 10.4c specifies failures of dynamic checks, either when the considered types have different heads, or when casting to or from the error type.

$$\Pi\text{-}\textsc{Germ}\ \frac{\Pi\, x\colon A.\, B \neq \mathrm{germ}_j \Pi \text{ for } j \geq i}{\langle ?_{\Box_i} \Leftarrow \Pi\, x\colon A.\, B \rangle\, f \rightharpoonup \langle ?_{\Box_i} \Leftarrow \mathrm{germ}_i\, \Pi \Leftarrow \Pi\, x\colon A.\, B \rangle\, f}$$

$$\textsc{List-Germ}\ \frac{\mathbf{Li}\,(A) \neq \mathrm{germ}_j I \text{ for } j \geq i}{\langle ?_{\Box_i} \Leftarrow \mathbf{Li}\,(A) \rangle\, t \rightharpoonup \langle ?_{\Box_i} \Leftarrow \mathrm{germ}_i\, \mathbf{Li} \Leftarrow \mathbf{Li}\,(A) \rangle\, t}$$

$$\textsc{Up-Down}\ \frac{\mathrm{germ}_i h \neq \mathrm{err}_{\Box_i}}{\langle X \Leftarrow ?_{\Box_i} \Leftarrow \mathrm{germ}_i h \rangle\, t \rightharpoonup \langle X \Leftarrow \mathrm{germ}_i h \rangle\, t} \qquad \textsc{Size-Err}\ \frac{\min\{j \mid \exists h \in \mathrm{Head}, \mathrm{germ}_j h = A\} > i}{\langle ?_{\Box_i} \Leftarrow A \rangle\, t \rightharpoonup \mathrm{err}_{?_{\Box_i}}}$$

Figure 10.4d. Casts and the unknown type

Finally, there are specific rules pertaining to casts to and from $?$, showcasing its behaviour as a universal type, given in Figure 10.4d. Rules Π-Germ and List-Germ decompose an upcast into $?$ into an upcast to a germ followed by an upcast from the germ to $?$. This decomposition of an upcast to $?$ into a series of "atomic" upcasts from a germ to $?$ is a consequence of the way the cast operation is implemented in Section 11.1, but similar decompositions appear *e.g.* in Siek et al. [Sie+15], where the equivalent of our germs are called ground types. The side conditions guarantee that this rule is used when no other applies.

[Sie+15]: Siek et al. (2015), *Refined Criteria for Gradual Typing*

Rule Up-Down erases the succession of an upcast to $?$ and a downcast from it. Note that in this rule the upcast $\langle ?_{\Box_i} \Leftarrow \mathrm{germ}_h i \rangle$ acts like a constructor for $?_{\Box_i}$, and $\langle X \Leftarrow ?_{\Box_i} \rangle$ as a destructor – a view reflected by the canonical and neutral forms for $?_{\Box}$ given in Figures 10.5b and 10.5c.[3]

3: In a simply-typed language such as GTLC [Sie+15], where there are no neutrals at the type level, casts from a germ or ground type to the unknown type are usually interpreted as tagged values [ST06]. Here, these correspond exactly to the canonical forms of $?_{\Box}$, but we also have to account for the many neutral forms that appear in open contexts.

Finally, Rule Size-Err corresponds to a peculiar kind of error, which only happens due to the presence of a type hierarchy: $?_{\Box_i}$ is only universal with respect to types at level i, and so a type might be of a level too high to fit

into it. To detect such a case, we check whether A is a germ for a level that is below i, and when not must raise an error.

10.1.2. Safety

Given the typing and reduction rules just define, we can already prove one of our main meta-theoretical properties: safety, for the three variants of CastCIC. The structure of the proof is very much the same as that of Part 'A Certified Kernel for Coq, in Coq' for PCUIC.

The crucial lemma is, as before, confluence:

Lemma 10.1. Confluence of CastCIC

If $t \to^\star t_1$ and $t \to^\star t_2$, then there exists t' such that $t_1 \to^\star t'$ and $t_2 \to^\star t'$.

Proof.

We follow again the Tait-Martin-Löf proof, as exposed by Takahashi [Tak95], extending the notion of parallel reduction from Section 7.3.1 to account for our additional reduction. The triangle property still holds, because as before there is no real critical pair between our rules – we carefully set them up to that effect!

□

[Tak95]: Takahashi (1995), *Parallel Reductions in λ-Calculus*

From this, exactly as in PCUIC we can obtain injectivity of type constructors, and thus finally, subject reduction follows. The only possibly surprising point, with respect to Chapter 7, is that we state it directly for the bidirectional system.

Theorem 10.2. Subject reduction for CastCIC

If $\Gamma \vdash t \triangleright T$ and $t \to^\star t'$ then $\Gamma \vdash t' \triangleleft T$.

Let us now turn to progress. To state progress, we must first extend our canonical forms, to encompass the three new term formers. This corresponds to giving intuition on what are the new canonical forms, and on "how" these new terms formers compute, in order to know when they are stuck, and thus give rise to a neutral form.

$$\frac{T \in \{\Box, \mathbf{Li}\,(A), ?_\Box, \mathrm{err}_\Box\}}{\mathrm{nm}\,?_T} \qquad \frac{\mathrm{ne}\,t}{\mathrm{ne}\,?_t}$$

$$\frac{T \in \{\Box, \mathbf{Li}\,(A), ?_\Box, \mathrm{err}_\Box\}}{\mathrm{nm}\,\mathrm{err}_T} \qquad \frac{\mathrm{ne}\,t}{\mathrm{ne}\,\mathrm{err}_t}$$

Figure 10.5a. Normal and neutral forms for ? and err

First, an error err_t or an unknown term $?_t$ is neutral when t is neutral, and is canonical when t is a canonical type – one of the canonical types of CIC, or the unknown or error types, but not a Π-type. This is detailed in Figure 10.5a. This is because exception-like terms reduce on Π-types – see Rule Π-Unk, and Pédrot and Tabareau [PT18].

$$\frac{}{\mathrm{nm}\ \langle ?_\Box \Leftarrow \mathrm{germ}_i h \rangle\, t}$$

Figure 10.5b. Cast as a canonical form of the unknown type

[PT18]: Pédrot et al. (2018), *Failure is Not an Option An Exceptional Type Theory*

Second come the canonical form for inhabitants of $?_\square$ (Figure 10.5b): these are upcasts from a germ, which can be seen as a term tagged with the head constructor of its type, in a matter reminiscent of actual implementations of dynamic typing using type tags. These canonical forms work as constructors for $?_\square$.

$$\frac{\mathrm{ne}\, S}{\mathrm{ne}\, \langle T \Leftarrow S \rangle\, t} \qquad \frac{\mathrm{ne}\, t}{\mathrm{ne}\, \langle T \Leftarrow ?_T \rangle\, t}$$

$$\frac{\mathrm{ne}\, T}{\mathrm{ne}\, \langle T \Leftarrow \square \rangle\, t} \qquad \frac{\mathrm{ne}\, T}{\mathrm{ne}\, \langle T \Leftarrow \Pi\, x\colon A.\ B \rangle\, t} \qquad \frac{\mathrm{ne}\, t}{\mathrm{ne}\, \langle \Pi\, x\colon A'.\ B' \Leftarrow \Pi\, x\colon A.\ B \rangle\, t}$$

$$\frac{\mathrm{ne}\, T}{\mathrm{ne}\, \langle T \Leftarrow \mathbf{Li}\,(A) \rangle\, t} \qquad \frac{\mathrm{ne}\, t}{\mathrm{ne}\, \langle \mathbf{Li}\,(A') \Leftarrow \mathbf{Li}\,(A) \rangle\, t}$$

Figure 10.5c. Neutral casts

Finally, the cast operation behaves as a destructor on the universe $\square$ – as if it were an inductive type of usual CIC. This destructor first scrutinizes the source type of the cast. This is why the cast is neutral as soon as its source type is neutral. When the source type reduces to a head constructor, there are two possibilities. Either that constructor is $?_\square$, in which case the cast scrutinizes whether its argument is a canonical form $\langle ?_\square \Leftarrow t \rangle\, \mathrm{germ}_i\, h$, and is neutral when this is not the case. In all other cases, it first scrutinizes the target type, so the cast is neutral when the target type is neutral. Finally, when both types have head constructors, the cast might still need its argument to be either a λ-abstraction or an inductive constructor to reduce.

Additionally, the notion of neutral terms naturally induces a weak-head reduction strategy, which reducing the (only) argument of the top-level destructor that is in a neutral position.

Equipped with the notion of canonical forms, we can state progress for CastCIC, and thus safety.

Theorem 10.3. Progress for CastCIC

> If t is a well-typed term of CastCIC, then either $\mathrm{nm}\, t$, or there is some t' such that $t \to^\star t'$.

Proof.

The proof is similar to that which has been sketched in Section 3.4: suppose a term is well-typed, and prove progress by induction on its typing derivation.

For the two cases of $?_T$ and err_T, this is direct. If T reduces, then the whole term reduces. In case it is neutral, the whole term is neutral. If it is a Π-type, the term reduces again – using *e.g.* Rule Π-Unk –. Finally, if it is another canonical type, then the whole term is canonical. Any other case is impossible, by typing.

For the cast term former, the proof still follows the same ideas. It is only complicated by the fact that in most cases all three sub-terms need to be canonical before the whole term can reduce. □

Theorem 10.4. Safety for CastCIC

> All three variants of CastCIC enjoy safety.

$$\frac{}{x \sim_\alpha x} \qquad \frac{}{\square_i \sim_\alpha \square_i} \qquad \frac{A \sim_\alpha A' \qquad t \sim_\alpha t'}{\lambda\, x{:}\,A.\, t \sim_\alpha \lambda\, x{:}\,A'.\, t'} \qquad \frac{A \sim_\alpha A' \qquad B \sim_\alpha B'}{\Pi\, x{:}\,A.\, B \sim_\alpha \Pi\, x{:}\,A'.\, B'} \qquad \frac{t \sim_\alpha t' \qquad u \sim_\alpha u'}{t\ u \sim_\alpha t'\ u'}$$

$$\frac{A \sim_\alpha A'}{\mathbf{Li}\,(A) \sim_\alpha \mathbf{Li}\,(A')} \qquad \frac{A \sim_\alpha A'}{\varepsilon_A \sim_\alpha \varepsilon_{A'}} \qquad \frac{A \sim_\alpha A' \qquad a \sim_\alpha a' \qquad l \sim_\alpha l'}{a ;;_A l \sim_\alpha a' ;;_{A'} l'}$$

$$\frac{s \sim_\alpha s' \qquad P \sim_\alpha P' \qquad b_\varepsilon \sim_\alpha b'_\varepsilon \qquad b_{;;} \sim_\alpha b'_{;;}}{\mathrm{ind}_{\mathbf{Li}}\,(s; z.P; b_\varepsilon, y_1.y_2.p_{y_2}.b_{;;}) \sim_\alpha \mathrm{ind}_{\mathbf{Li}}\,(s'; z.P'; b'_\varepsilon, y_1.y_2.p_{y_2}.b'_{;;})} \qquad \frac{t \sim_\alpha t'}{t \sim_\alpha \langle B' \Leftarrow A' \rangle\, t'} \qquad \frac{t \sim_\alpha t'}{\langle B \Leftarrow A \rangle\, t \sim_\alpha t'}$$

$$\frac{}{t \sim_\alpha ?_{T'}} \qquad \frac{}{?_T \sim_\alpha t}$$

Figure 10.6. CastCIC: α-consistency

10.2. Bidirectional Elaboration: from GCIC to CastCIC

Now that CastCIC has been described, let us move on to GCIC. The typing judgement of GCIC is *defined* by an elaboration judgement from GCIC to CastCIC, based upon that of Part 'Bidirectional Calculus of Inductive Constructions', but augmenting all judgements with an extra output: the elaborated CastCIC term. This definition of typing using elaboration is required because of the intricate interdependency between typing and reduction exposed in Section 9.6.

10.2.1. System definition

Syntax. The syntax of *GCIC*[4] extends that of CIC with a single new term constructor $?_i$, where i is a universe level. From a user perspective, one is not given direct access to the failure and cast primitives, those only arise through uses of $?$.

4: We use purple for terms of GCIC. To maintain a distinction in the absence of colours, we also use tildes – like so $\tilde{t}$ – for terms in GCIC in expressions mixing both source and target terms.

Consistent conversion. Before we can describe typing, we should focus on conversion. Indeed, to account for the imprecision introduced by the unknown term, elaboration employs consistent conversion to compare CastCIC terms, rather than usual conversion relation.

Definition 10.5. Consistent conversion

> Two CastCIC terms are *α-consistent*, written $\sim_\alpha$, if they are in the relation defined by the inductive rules of Figure 10.6.
>
> Two terms are *consistently convertible* – or simply *consistent*, noted $s \sim t$, if and only if there exists s' and t' such that $s \to^\star s'$, $t \to^\star t'$ and $s' \sim_\alpha t'$.

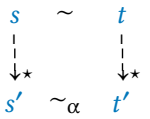

Figure 10.7. Consistent conversion, as a diagram

Thus, α-consistency is an extension of α-equality that takes imprecision into account. Apart from the standard rules making $?$ consistent with any

term, α-consistency optimistically ignores casts, and does not consider errors to be consistent with themselves. The first point is to prevent casts inserted by the elaboration from disrupting valid conversions, typically between static terms. The second is guided by the idea that if errors are encountered at elaboration already, the term cannot be well-behaved, so it must be rejected as early as possible, and we should avoid typing it. The consistency relation is then built upon α-consistency in a way totally similar to how algorithmic conversion in Figure 3.6a is built upon α-equality.

It is very important at this point to extend algorithmic conversion, rather than declarative conversion, because we do *not* want consistency to be transitive, since we wish to have $t \sim ?$ for any t, which would turn $\sim$ into the full relation if it were to be transitive. Thus, we must extend a relation where transitivity is not baked in. Also note that this formulation of consistent conversion makes no assumption of normalization, and is therefore usable as such in the non-normalizing $\text{GCIC}^{\mathcal{G}}$.

An important property of consistent conversion, and a necessary condition for conservativity of GCIC with respect to CIC, is that it corresponds to conversion on static terms.

Proposition 10.6. Properties of consistent conversion

- Two static terms are consistently convertible if and only if they are convertible in CIC.
- If s and t have a normal form, then $s \sim t$ is decidable.

Proof.

For the first point, first remark that α-consistency between static terms corresponds to α-equality of terms. Thus, and because the reduction of static terms in CastCIC is the same as the reduction of CIC, two consistent static terms must reduce to α-equal terms, which in turn implies that they are convertible. Conversely, two convertible terms of CIC have α-equal reducts, which are also α-consistent.

For the second point, if s and t are normalizing, they have a finite number of reducts. Thus, to decide their consistency it is sufficient to check each pair of reducts for the decidable α-consistency. We conjecture that the more reasonable algorithm which is used in practice in *e.g.* Coq for deciding conversion, and relies on iterated weak-head normalization, can be adapted to decide consistency. This would however need somewhat subtle proofs, in the vein of those we use in order to prove the DGG. □

Elaboration. Elaboration from GCIC to CastCIC closely follows the bidirectional presentation of CIC given in Part 'Bidirectional Calculus of Inductive Constructions' for most rules, simply carrying around the extra elaborated term: see Figure 10.8a. Note that only the subject of the judgement is a source term in GCIC; inputs – that have already been elaborated –, as well as outputs – that are to be constructed –, are target terms in CastCIC. In particular, the extra elaborated term in CastCIC is an output, in all judgements.

$$\textsc{Unk}\ \frac{}{\Gamma \vdash ?_i \rightsquigarrow ?_{?_{\square_i}} \triangleright ?_{\square_i}}$$

Figure 10.8b. Type-directed elaboration for ?

Next comes Rule Unk: $?_i$ is elaborated to $?_{?_{\square_i}}$, the least precise term of the least precise type of the whole universe $\square_i$. This avoids unneeded type annotations on ? in GCIC. Instead, the context is responsible for inserting

$$\textsc{Var}\ \frac{(x:T) \in \Gamma}{\Gamma \vdash x \rightsquigarrow x \triangleright T} \qquad \textsc{Univ}\ \frac{}{\Gamma \vdash \Box_i \rightsquigarrow \Box_i \triangleright \Box_{i+1}} \qquad \Pi\textsc{Ty}\ \frac{\Gamma \vdash \tilde{A} \rightsquigarrow A \triangleright_\Box \Box_i \qquad \Gamma, x:A \vdash \tilde{B} \rightsquigarrow B \triangleright_\Box \Box_j}{\Gamma \vdash \Pi\, x:\tilde{A}.\, \tilde{B} \rightsquigarrow \Pi\, x:A.\, B \triangleright \Box_{s_\Pi(i,j)}}$$

$$\textsc{Abs}\ \frac{\Gamma \vdash \tilde{A} \rightsquigarrow A \triangleright_\Box \Box \qquad \Gamma, x:A \vdash \tilde{t} \rightsquigarrow t \triangleright B}{\Gamma \vdash \lambda\, x:\tilde{A}.\, \tilde{t} \rightsquigarrow \lambda\, x:A.\, t \triangleright \Pi\, x:A.\, B} \qquad \textsc{App}\ \frac{\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright_\Pi \Pi\, x:A.\, B \qquad \Gamma \vdash \tilde{u} \rightsquigarrow u \triangleleft A}{\Gamma \vdash \tilde{t}\ \tilde{u} \rightsquigarrow t\ u \triangleright B[x := u]}$$

$$\textsc{ListTy}\ \frac{\Gamma \vdash \tilde{A} \rightsquigarrow A \triangleright_\Box \Box_i}{\Gamma \vdash \mathbf{Li}\,(\tilde{A}) \rightsquigarrow \mathbf{Li}\,(A) \triangleright \Box_i} \qquad \textsc{Nil}\ \frac{\Gamma \vdash \tilde{A} \rightsquigarrow A \triangleright_\Box \Box}{\Gamma \vdash \varepsilon_{\tilde{A}} \rightsquigarrow \varepsilon_A \triangleright \mathbf{Li}\,(A)}$$

$$\textsc{Cons}\ \frac{\Gamma \vdash \tilde{A} \rightsquigarrow A \triangleright_\Box \Box \qquad \Gamma \vdash \tilde{a} \rightsquigarrow a \triangleleft A \qquad \Gamma \vdash \tilde{l} \rightsquigarrow l \triangleleft \mathbf{Li}\,(A)}{\Gamma \vdash \tilde{a} ;;_{\tilde{A}} \tilde{l} \rightsquigarrow a ;;_A l \triangleright \mathbf{Li}\,(A)}$$

$$\textsc{Fix}\ \frac{\begin{array}{c}\Gamma \vdash \tilde{s} \rightsquigarrow s \triangleright_{\mathbf{Li}} \mathbf{Li}\,(A) \qquad \Gamma, z:\mathbf{Li}\,(A) \vdash \tilde{P} \rightsquigarrow P \triangleright_\Box \Box \\ \Gamma \vdash \tilde{b}_\varepsilon \rightsquigarrow b_\varepsilon \triangleleft P[z := \varepsilon] \qquad \Gamma, y_1:A, y_2:\mathbf{Li}\,(A), p_{y_2}:P[z := y_2] \vdash \tilde{b}_{;;} \rightsquigarrow b_{;;} \triangleleft P[z := y_1 ;;_A y_2]\end{array}}{\Gamma \vdash \mathrm{ind}_{\mathbf{Li}}\,(\tilde{s}; z.\tilde{P}; \tilde{b}_\varepsilon, y_1.y_2.p_{y_2}.\tilde{b}_{;;}) \rightsquigarrow \mathrm{ind}_{\mathbf{Li}}\,(s; z.P; b_\varepsilon, y_1.y_2.p_{y_2}.b_{;;}) \triangleright P[z := s]}$$

Figure 10.8a. Type-directed elaboration of GCIC: static fragment

the appropriate cast, *e.g.* $? :: T$ elaborates to a term reducing to $?_T$. We do not drop annotations altogether because we wish to keep the property that any well-formed term should *infer* a type, not just check. Thus, we must be able to infer a type for $?$. The obvious choice is to have $?$ infer $?$, but this $?$ is a term of CastCIC, and thus needs a type index. Because this $?$ is used as a type, this index must be $\Box$, and the universe level of the source $?$ is there to give us the level of this $\Box$. In a real system like Coq, this should be handled by typical ambiguity, alleviating the user from the need to give any annotations when using $?$.

The most salient feature of elaboration is however the insertion of casts that mediate between merely consistent but not convertible types, which is done in the checking and constrained inference judgements, see Figure 10.8c. They of course are needed in Rule Check, where the terms are compared using consistency. But this is not enough: casts also appear in the newly-introduced Rules Inf-Univ?, Inf-Prod? and Inf-List? for constrained inference, where the type $?_{\Box_i}$ is replaced by the least precise type of the appropriate universe level having the constrained head constructor, which is exactly what the germ function computes. Note that in the case of Inf-Univ? we could have replaced $\Box_i$ with $\mathrm{germ}_{i+1}\Box_i$ to make the presentation more uniform with respect to the other two rules. The role of these three rules is to ensure that a term of type $?_{\Box_i}$ can be used as a function, or as a scrutinee of a match, by giving a way to derive constrained inference for such a term.

It is interesting to observe that the rules for constrained elaboration in a gradual setting bear a close resemblance with those described by Cimini and Siek [CS16, Section 3.3], where a matching operator is introduced to verify that an output type can fit into a certain type constructor – either by having that type constructor as head symbol or by virtue of being $?$. Such a form of matching was already present in our static, bidirectional system, because of the presence of reduction in types. In a way, both Cimini and Siek [CS16] and Part 'Bidirectional Calculus of Inductive Constructions'

[CS16]: Cimini et al. (2016), *The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems*

$$\textsc{Check}\ \frac{\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright T \qquad T \sim S}{\Gamma \vdash \tilde{t} \rightsquigarrow \langle S \Leftarrow T \rangle t \triangleleft S}$$

$$\textsc{Inf-Univ}\ \frac{\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright T \qquad T \to^\star \square_i}{\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright_\square \square_i} \qquad \textsc{Inf-Univ?}\ \frac{\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright T \qquad T \to^\star ?_{\square_{i+1}}}{\Gamma \vdash \tilde{t} \rightsquigarrow \langle \square_i \Leftarrow T \rangle t \triangleright_\square \square_i}$$

$$\textsc{Inf-Prod}\ \frac{\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright T \qquad T \to^\star \Pi x{:}A.\, B}{\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright_\Pi \Pi x{:}A.\, B} \qquad \textsc{Inf-Prod?}\ \frac{\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright T \qquad T \to^\star ?_{\square_i} \qquad \mathrm{c}_\Pi(i) \geq 0}{\Gamma \vdash \tilde{t} \rightsquigarrow \langle \mathrm{germ}_i\, \Pi \Leftarrow T \rangle t \triangleright_\Pi \mathrm{germ}_i \Pi}$$

$$\textsc{Inf-List}\ \frac{\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright T \qquad T \to^\star \mathbf{Li}(A)}{\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright_{\mathbf{Li}} \mathbf{Li}(A)} \qquad \textsc{Inf-List?}\ \frac{\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright T \qquad T \to^\star ?_{\square_i}}{\Gamma \vdash \tilde{t} \rightsquigarrow \langle \mathrm{germ}_i\, \mathbf{Li} \Leftarrow T \rangle t \triangleright_{\mathbf{Li}} \mathrm{germ}_i\, \mathbf{Li}}$$

Figure 10.8c. Type-directed elaboration for GCIC: constrained judgements

have the same need of separating the inferred type from operations on it to recover its head constructor, and our mixing of both computation and gradual typing makes that need even clearer.

10.2.2. Direct properties

Let us establish already some important properties of elaboration that we can prove at this stage. First, elaboration is *correct*, insofar as it always produces well-typed CastCIC terms.

Theorem 10.7. Correctness of elaboration

> Elaboration produces well-typed terms in a well-formed context. Namely, given Γ such that $\vdash \Gamma$, we have that if $\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright T$, then $\Gamma \vdash t \triangleright T$.

Proof.

The proof is by induction on the elaboration derivation, mutually with similar properties for all elaboration judgements. It resembles a lot the correctness proof for bidirectional typing (Theorem 4.3). In particular, for checking, we have an extra hypothesis that the given type is well-formed, since it is an input that should already have been typed.

Because the bidirectional typing rules of CIC are very close to the GCIC-to-CastCIC elaboration rules, the induction is mostly straightforward. Let us point however that once again the careful design of the elaboration rules to respect McBride's discipline – see Section 4.1 – is crucial for the proof to go through.

The main novel points to consider is the rules where a cast is inserted. For these, we rely on the validity property – an inferred type is always itself well-typed – to ensure that the domain of inserted casts is well-typed, and thus that the casts can be typed. □

Next come the more "algorithmic" properties: elaboration is decidable, and outputs are unique – up to conversion if no strategy is fixed.

Theorem 10.8. Decidability of elaboration

> The elaboration relation of Figures 10.8a to 10.8c is decidable in GCIC$^{\mathcal{N}}$ and GCIC$^{\uparrow}$. It is semi-decidable in GCIC$^{\mathcal{G}}$.

Proof.

As the elaboration rules are completely syntax-directed, they immediately translate to an algorithm for elaboration. Coupled with decidability of consistency (Proposition 10.6), this makes elaboration decidable whenever $\rightarrow^\star$ is normalizing; when $\rightarrow^\star$ is not normalizing, the elaboration algorithm might diverge, resulting in only semi-decidability of typing – as in *e.g.* Dependent Haskell [Eis16].

[Eis16]: Eisenberg (2016), *Dependent Types in Haskell: Theory and Practice*

□

As was the case for bidirectional typing – Theorems 4.4 and 4.6 – there are two versions of uniqueness: one is uniqueness up to conversion, in case full reduction is used. The second is a strengthening if a weak-head reduction strategy is imposed for reduction.

Theorem 10.9. Uniqueness of elaboration – Full reduction

> Elaborated terms are convertible: If $\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright T$ and $\Gamma \vdash \tilde{t} \rightsquigarrow t' \triangleright T'$, then $t \cong t'$ and $T \cong T'$.

Theorem 10.10. Uniqueness of elaboration – Weak-head reduction

> If in Figure 10.8c, full reduction $\rightarrow^\star$ is replaced by weak-head reduction, then elaborated terms are unique: given Γ and $\tilde{t}$, there is at most one t and one T such that $\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright T$.

Proof.

Like for Theorems 4.4 and 4.6, those are proven mutually by induction on the typing derivation. Again, the main argument is that there is always at most one rule that can apply to get a typing conclusion for a given term.

This is true for all inference statements because there is exactly one inference rule for each term constructor, and for checking because there is only one rule to derive checking. In those cases simply combining the hypothesis of uniqueness is enough.

For $\triangleright_\Pi$, by confluence of CastCIC the inferred type cannot at the same time reduce to $?_\square$ and $\Pi\, x{:}\, A.\, B$, because those do not have a common reduct. Thus, only one of Rule Inf-Prod and Rule Inf-Prod? can apply. It is enough to conclude for Theorem 10.9, because reducts of convertible types are still convertible. For Theorem 10.10 the deterministic reduction strategy ensures that the inferred type is unique, rather than unique up to conversion.

The reasoning is similar for the other constrained inference judgements.

□

10.2.3. Illustration: back to Ω

Now that GCIC has been entirely presented, let us come back to the important example of Ω, and explain in detail the behaviour described in Section 9.6.1 for the three GCIC variants.

Recall that Ω is the term $\delta\,\delta$, with $\delta \coloneqq \lambda\, x\colon ?_{i+1}.\, x\, x$. We leave out the casts present in Section 9.6, knowing that they will be introduced by elaboration. We also use $?$ at level $i{+}1$, because $?_{i+1}$, when elaborated as a type, becomes $T_i \coloneqq \langle \square_i \Leftarrow ?_{\square_{i+1}} \rangle\, ?_{?_{\square_{i+1}}}$, such that $T_i \to^\star ?_{\square_i}$. For the rest of this section, we write $?_j$ instead of $?_{\square_j}$ to avoid stacked indices and ease readability.

If $i = 0$ the elaboration of δ – and thus of Ω – fails in GCIC$^\uparrow$ and GCIC$^{\mathcal{N}}$, because the inferred type for x is T_0, which reduces to $?_0$. Then, because $\mathrm{c}_\Pi(0) = -1 < 0$ in both GCIC$^\uparrow$ and GCIC$^{\mathcal{N}}$, Rule Inf-Prod? does not apply and δ is deemed ill-typed, and so is Ω.

Otherwise, if $i > 0$ or we are considering GCIC$^{\mathcal{G}}$, δ can be elaborated, and we have

$$\cdot \vdash \delta \rightsquigarrow \lambda\, x\colon T_i.\ (\langle \mathrm{germ}_i \Pi \Leftarrow T_i \rangle\, x)\ \left(\langle ?_{\mathrm{c}_\Pi(i)} \Leftarrow T_i \rangle\, x\right) \triangleright T_i \to ?_{\mathrm{c}_\Pi(i)}$$

From this, we get that Ω also elaborates, namely

$$\cdot \vdash \Omega \rightsquigarrow \delta'\ \left(\langle T \Leftarrow T \to ?_{\mathrm{c}_\Pi(i)} \rangle\, \delta'\right) \triangleright ?_{\mathrm{c}_\Pi(i)}$$

with δ' the elaboration of δ above. Let us now look at the reduction behaviour of this elaborated term Ω' in the three systems: it reduces seamlessly when $\mathrm{c}_\Pi(i) = i$ (CastCIC$^{\mathcal{G}}$), while having $\mathrm{c}_\Pi(i) < i$ makes it fail (CastCIC$^\uparrow$ and CastCIC$^{\mathcal{N}}$).

The reduction of Ω' in CastCIC$^{\mathcal{G}}$ is as follows:

$$\begin{aligned}
& \Omega' \\
\to^\star\ & (\lambda\, x\colon ?_i.\, (\langle ?_i \to ?_i \Leftarrow T \rangle\, x)\ (\langle ?_i \Leftarrow T \rangle\, x))\ (\langle T \Leftarrow T \to ?_i \rangle\, \delta') \\
\to^\star\ & (\lambda\, x\colon ?_i.\, (\langle ?_i \to ?_i \Leftarrow ?_i \rangle\, x)\ (\langle ?_i \Leftarrow ?_i \rangle\, x))\, (\langle ?_i \Leftarrow ?_i \to ?_i \rangle\, \delta') \\
\to^\star\ & (\langle ?_i \to ?_i \Leftarrow ?_i \Leftarrow ?_i \to ?_i \rangle\, \delta')\ (\langle ?_i \Leftarrow ?_i \Leftarrow ?_i \to ?_i \rangle\, \delta') \\
\to^\star\ & (\langle ?_i \to ?_i \Leftarrow ?_i \to ?_i \rangle\, \delta')\ (\langle ?_i \Leftarrow ?_i \to ?_i \rangle\, \delta') \\
\to^\star\ & (\lambda\, x\colon ?_i.\, \langle ?_i \Leftarrow ?_i \rangle\, ((\langle ?_i \to ?_i \Leftarrow ?_i \rangle\, x)\, (\langle ?_i \Leftarrow ?_i \Leftarrow ?_i \rangle\, x))) \\
& \qquad\qquad (\langle ?_i \Leftarrow ?_i \to ?_i \rangle\, \delta')
\end{aligned}$$

The first step is the identity, simply replacing Ω', $\mathrm{c}_\Pi(i)$ and the first occurrence of δ' by their definitions. The second reduces T to $?_i$. In the third, the cast δ' is substituted for x by a β-step. Casts are finally simplified using Rule Up-Down and Rule Π-Π. At that point, the reduction has almost looped back to the second step, apart from the casts $\langle ?_i \Leftarrow ?_i \rangle$ in the first occurrence of δ', which will simply accumulate through reduction, but without hindering divergence.

On the contrary, the normalizing variants have $\mathrm{c}_\Pi(i) < i$, and thus share

the following reduction path:

$$
\begin{array}{lll}
\Omega' & \to^\star & \delta''\,(\langle ?_{i-1} \Leftarrow ?_i \Leftarrow ?_i \to ?_{i-1}\rangle\ \delta') \\
& & \text{where } \delta'' \text{ is } (\langle ?_{i-1} \to ?_{i-1} \Leftarrow ?_i \Leftarrow ?_i \to ?_{i-1}\rangle\ \delta') \\
& \to^\star & \delta''\,(\langle ?_{i-1} \Leftarrow ?_{i-1} \to ?_{i-1} \Leftarrow ?_i \to ?_{i-1}\rangle\ \delta') \\
& \to^\star & \delta''\,\mathrm{err}_{?_{i-1}} \\
& \to^\star & (\langle ?_{i-1} \to ?_{i-1} \Leftarrow ?_{i-1} \to ?_{i-1} \Leftarrow ?_i \to ?_{i-1}\rangle\ \delta')\,\mathrm{err}_{?_{i-1}} \\
& \to^\star & (\lambda\, x{:}?_{i-1}.\,\langle ?_{i-1} \Leftarrow ?_{i-1} \Leftarrow ?_{i-1}\rangle\,(x''\,(\langle ?_{i-1} \Leftarrow ?_i\rangle\ x'))\,)\,\mathrm{err}_{?_{i-1}} \\
& & \text{where } x' \text{ is } \langle ?_i \Leftarrow ?_{i-1} \Leftarrow ?_{i-1}\rangle\ x \text{ and } x'' \text{ is } \langle ?_{i-1} \to ?_{i-1} \Leftarrow ?_i\rangle\ x' \\
& \to^\star & \langle ?_{i-1} \Leftarrow ?_{i-1} \Leftarrow ?_{i-1}\rangle(\mathrm{err}_{?_{i-1}\to ?_{i-1}}\,\mathrm{err}_{?_{i-1}}) \\
& \to^\star & \mathrm{err}_{?_{i-1}}
\end{array}
$$

The first step corresponds to the first three above, the only difference being the value of $c_\Pi\,(i)$. The reductions however differ in the next step because $?_i \to ?_{i-1} \neq \mathrm{germ}_i\Pi$, so Rule Π-Germ applies before Rule Up-Down. For the third step, note that $?_{i-1} \to ?_{i-1} = \mathrm{germ}_i\,\Pi$, so that Rule Size-Err applies in the rightmost sequence of casts. The last three steps of reduction then propagate the error by first using Rule Π-Germ, Rule Up-Down and Rule Π-Π, then the β-rule, and finally Rule Down-Err, Rule Π-Err and a last β step. At a high-level, the error can be seen as a dynamic universe inconsistency, triggered by the invalid downcast $\langle ?_{i-1} \Leftarrow ?_i\rangle$ highlighted on the first line.

10.3. Precision is a Simulation for Reduction

Establishing elaboration graduality – the formulation of the static gradual guarantee SGG in our setting – is no small feat, as it requires properties about computations in CastCIC that amount to the dynamic gradual guarantee (DGG). Indeed, to handle the typing rules for checking and constrained inference, it is necessary to know how consistency and reduction evolve as a type becomes less precise.

As already explained in Section 9.6, we cannot directly prove graduality for a syntactic notion of precision. However, we can still show that such a syntactic precision is a simulation for reduction. While weaker than graduality, this property implies the DGG, and suffices to conclude that graduality of elaboration holds.

The purpose of this section is to establish this property. Our proof is partly inspired by the proof of DGG by Siek et al. [Sie+15] for the simply-typed lambda calculus.[5] We however have to adapt to the much higher complexity of CIC compared to STLC. In particular, the presence of computation in the domain and codomain of casts is quite subtle to tame, as we must in general reduce types in a cast before we can reduce the cast itself.[6]

Technically, we need to distinguish between two notions of precision, one for GCIC and one for CastCIC: *syntactic precision*, on terms in GCIC, which corresponds to the usual syntactic precision of gradual typing, such as that of Siek et al. [Sie+15]; and *structural precision* on terms in CastCIC, which corresponds to syntactic precision, together with a proper account of casts. In this section, we concentrate on properties of structural precision – in CastCIC. We only state and discuss the various lemmas and theorems on a rather high level, and refer the reader to Lennon-Bertrand et al. [Len+22, Appendix B] for the detailed proofs.

[Sie+15]: Siek et al. (2015), *Refined Criteria for Gradual Typing*

5: Lemma 7 in Siek et al. [Sie+15] is similar to our Theorem 10.16, and Figures 10.9a to 10.9c draws from their Fig. 9, especially for Rule Cast-R and Rule Cast-L. Also, while we do not make them explicit, Lemmas 8, 10 and 11 also appear in our proofs.

6: Thus, while Lemmas 10.14 and 10.15 correspond roughly to Lemma 9 in Siek et al. [Sie+15], Lemmas 10.12 and 10.13 are completely novel.

[Len+22]: Lennon-Bertrand et al. (2022), *Gradualizing the Calculus of Inductive Constructions*

10.3.1. Structural precision for CastCIC

As emphasized already, the key property we want to establish is that precision is a simulation for reduction, *i.e.* that less precise terms reduce at least as well as more precise ones. This property guides the quite involved definition we are about to give for structural precision: it is rigid enough to give the induction hypotheses needed to prove the simulation, while being lax enough to be a consequence of syntactic precision after elaboration, which is the key point to establish elaboration graduality (Theorem 10.23), our equivalent of the static gradual guarantee.

Similarly to α-consistency, precision can ignore some casts, in order to handle the cases when those might appear or disappear in one term but not the other during reduction. But in order to control what casts can be ignored, we impose some restriction on the types involved. In particular, we want to ensure that ignored casts would not have raised an error: *e.g.* we want to prevent $0 \sqsubseteq_\alpha \langle \mathbf{B} \Leftarrow \mathbf{N} \rangle\, 0$. Thus, the definition of structural precision relies on typing, and to do this we need to record the contexts of the two compared terms. To denote such contexts where each variable is given two types, we use double-struck letters, writing $\mathbb{\Gamma}, x\colon A \mid A'$ for context extensions. We use $\mathbb{\Gamma}_{\cdot i}$ for projections, *i.e.* $(\mathbb{\Gamma}, x\colon A \mid A')_{\cdot 1} \coloneqq \mathbb{\Gamma}_{\cdot 1}, x\colon A$, and write $\Gamma \mid \Gamma'$ for the converse pairing operation.

$$\textsc{Univ-Diag}\ \frac{}{\mathbb{\Gamma} \vdash \square_i \sqsubseteq_\alpha \square_i} \qquad \Pi\textsc{-Diag}\ \frac{\mathbb{\Gamma} \vdash A \sqsubseteq_\alpha A' \qquad \mathbb{\Gamma}, x\colon A \mid A' \vdash B \sqsubseteq_\alpha B'}{\mathbb{\Gamma} \vdash \Pi\, x\colon A.\, B \sqsubseteq_\alpha \Pi\, x\colon A'.\, B'}$$

$$\textsc{Abs-Diag}\ \frac{\mathbb{\Gamma} \vdash A \sqsubseteq_\to A' \qquad \mathbb{\Gamma}, x\colon A \mid A' \vdash t \sqsubseteq_\alpha t'}{\mathbb{\Gamma} \vdash \lambda\, x\colon A.t \sqsubseteq_\alpha \lambda\, x\colon A'.t'} \qquad \textsc{App-Diag}\ \frac{\mathbb{\Gamma} \vdash t \sqsubseteq_\alpha t' \qquad \mathbb{\Gamma} \vdash u \sqsubseteq_\alpha u'}{\mathbb{\Gamma} \vdash t\ u \sqsubseteq_\alpha t'\ u'}$$

$$\textsc{Var-Diag}\ \frac{}{\mathbb{\Gamma} \vdash x \sqsubseteq_\alpha x} \qquad \textsc{List-Diag}\ \frac{\mathbb{\Gamma} \vdash A \sqsubseteq_\alpha A'}{\mathbb{\Gamma} \vdash \mathbf{Li}\,(A) \sqsubseteq_\alpha \mathbf{Li}\,(A')} \qquad \textsc{Nil-Diag}\ \frac{\mathbb{\Gamma} \vdash A \sqsubseteq_\alpha A'}{\mathbb{\Gamma} \vdash \varepsilon_A \sqsubseteq_\alpha \varepsilon_{A'}}$$

$$\textsc{Cons-Diag}\ \frac{\mathbb{\Gamma} \vdash A \sqsubseteq_\alpha A' \qquad \mathbb{\Gamma} \vdash a \sqsubseteq_\alpha a' \qquad \mathbb{\Gamma} \vdash l \sqsubseteq_\alpha l'}{\mathbb{\Gamma} \vdash a ;;_A l \sqsubseteq_\alpha a' ;;_{A'} l'}$$

$$\textsc{Ind-Diag}\ \frac{\begin{array}{c}\mathbb{\Gamma} \vdash s \sqsubseteq_\alpha s' \qquad \mathbb{\Gamma}_{\cdot 1} \vdash s \triangleright_{\mathbf{Li}} \mathbf{Li}\,(A) \qquad \mathbb{\Gamma}_{\cdot 2} \vdash s' \triangleright_{\mathbf{Li}} \mathbf{Li}\,(A') \qquad \mathbb{\Gamma}, z\colon \mathbf{Li}\,(A) \mid \mathbf{Li}\,(A') \vdash P \sqsubseteq_\alpha P' \\ \mathbb{\Gamma} \vdash b_\varepsilon \sqsubseteq_\alpha b'_\varepsilon \qquad \mathbb{\Gamma}, y_1\colon A \mid A', y_2\colon \mathbf{Li}\,(A) \mid \mathbf{Li}\,(A'), p_{y_2}\colon P[z \coloneqq y_2] \mid P'[z \coloneqq y_2] \vdash b_{;;} \sqsubseteq_\alpha b'_{;;}\end{array}}{\mathbb{\Gamma} \vdash \mathrm{ind}_{\mathbf{Li}}\,(s; z.P; b_\varepsilon, y_1.y_2.p_{y_2}.b_{;;}) \sqsubseteq_\alpha \mathrm{ind}_{\mathbf{Li}}\,(s'; z.P'; b'_\varepsilon, y_1.y_2.p_{y_2}.b'_{;;})}$$

$$\textsc{Cast-Diag}\ \frac{\mathbb{\Gamma} \vdash A \sqsubseteq_\alpha A' \qquad \mathbb{\Gamma} \vdash B \sqsubseteq_\alpha B' \qquad \mathbb{\Gamma} \vdash t \sqsubseteq_\alpha t'}{\mathbb{\Gamma} \vdash \langle B \Leftarrow A \rangle\, t \sqsubseteq_\alpha \langle B' \Leftarrow A' \rangle\, t'}$$

Figure 10.9a. Structural precision in CastCIC, diagonal rules

Definition 10.11. Structural and definitional precision in CastCIC

> *Structural precision*, denoted $\mathbb{\Gamma} \vdash t \sqsubseteq_\alpha t'$, is defined in Figures 10.9a to 10.9c, mutually with *definitional precision*, denoted $\mathbb{\Gamma} \vdash t \sqsubseteq_\to t'$ and defined in Figure 10.9d.
> We write $\Gamma \sqsubseteq_\alpha \Gamma'$ and $\Gamma \sqsubseteq_\to \Gamma'$ for the pointwise extensions of those to contexts.

Let us now detail the rules defining structural precision. Diagonal rules

of Figure 10.9a correspond to congruence closure, and there is not much to be said here. The only subtlety is with Rule Ind-Diag, where typing assumptions are needed to provide us with the contexts used to compare the predicates.

$$\textsc{Unk}\ \frac{\Gamma.1 \vdash t \triangleright T \qquad \Gamma \vdash T \sqsubseteq_{\to} T'}{\Gamma \vdash t \sqsubseteq_{\alpha} ?_{T'}} \qquad \textsc{Unk-Univ}\ \frac{\Gamma.1 \vdash A \triangleright_{\square} \square_i \qquad i \leq j}{\Gamma \vdash A \sqsubseteq_{\alpha} ?_{\square_j}}$$

$$\textsc{Err}\ \frac{\Gamma.2 \vdash t' \triangleright T' \qquad \Gamma \vdash T \sqsubseteq_{\to} T'}{\Gamma \vdash \mathrm{err}_T \sqsubseteq_{\alpha} t'} \qquad \textsc{Err-}\lambda\ \frac{\Gamma.1 \vdash t' \triangleright_{\Pi} \Pi\, x{:}A'.B' \qquad \Gamma \vdash \Pi\, x{:}A.\, B \sqsubseteq_{\to} \Pi\, x{:}A'.\, B'}{\Gamma \vdash \lambda\, x{:}A.\, \mathrm{err}_B \sqsubseteq_{\alpha} t'}$$

Figure 10.9b. Structural precision in CastCIC, unknown and error

More interesting are the non-diagonal rules. First, those for ? and err. The unknown $?_T$ is greater than any term of the "right type". This incorporates loss of precision – Rule Unk –, and accommodates for a small bit of cumulativity per Rule Unk-Univ. This is needed because of technical reasons linked with the possibility to form products between types at different levels. On the contrary, the error is smaller than any term – Rule Err –, even in its extended form on Π-types – Rule Err-λ –, with a typing premise similar to that of Rule Unk.

$$\textsc{Cast-R}\ \frac{\Gamma.1 \vdash t \vdash T \triangleright \qquad \Gamma \vdash T \sqsubseteq_{\to} A' \qquad \Gamma \vdash T \sqsubseteq_{\to} B' \qquad \Gamma \vdash t \sqsubseteq_{\alpha} t'}{\Gamma \vdash t \sqsubseteq_{\alpha} \langle B' \Leftarrow A' \rangle\, t'}$$

$$\textsc{Cast-L}\ \frac{\Gamma.2 \vdash t' \vdash T' \triangleright \qquad \Gamma \vdash A \sqsubseteq_{\to} T' \qquad \Gamma \vdash B \sqsubseteq_{\to} T' \qquad \Gamma \vdash t \sqsubseteq_{\alpha} t'}{\Gamma \vdash \langle B \Leftarrow A \rangle\, t \sqsubseteq_{\alpha} t'}$$

Figure 10.9c. Structural precision in CastCIC, cast rules

Finally, casts on the right-hand side can be ignored as long as they are performed on types that are *less* precise than the type of the term on the left – Rule Cast-R. Dually, casts on the left-hand side can be ignored as long as they are performed on types that are *more* precise than the type of the term on the right – Rule Cast-L.

$$\frac{\Gamma \vdash t \sqsubseteq_{\alpha} t'}{\Gamma \vdash t \sqsubseteq_{\to} t'} \qquad \frac{\Gamma \vdash s \sqsubseteq_{\to} t' \qquad t \to^1 s}{\Gamma \vdash t \sqsubseteq_{\to} t'} \qquad \frac{\Gamma \vdash t \sqsubseteq_{\to} s' \qquad t' \to^1 s'}{\Gamma \vdash t \sqsubseteq_{\to} t'}$$

Figure 10.9d. Definitional precision in CastCIC

As for definitional precision, $\Gamma \vdash t \sqsubseteq_{\to} t'$ is defined in a stepwise way – to ease proofs by induction –, but is equivalent to the existence of s and s' such that $t \to^\star s$, $t' \to^\star s'$ and $\Gamma \vdash s \sqsubseteq_{\alpha} s'$. The situation is the same as for consistency – respectively algorithmic conversion –, which is the closure by reduction of α-consistency – respectively α-equality. However, here definitional precision is also *used* in the definition of structural precision, in order to permit computation in types – recall that in a dependently-typed setting the two types involved in a cast may need to reduce before the cast itself can reduce – and thus the two notions must be mutually defined.

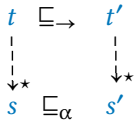

Figure 10.10. Definitional precision, diagrammatically

10.3.2. Catch-up lemmas

The fact that structural precision is a simulation relies on a series of lemmas, which constitute the technical core of this whole chapter. They all have the same form: under the assumption that a term t' is less precise than a term t which is a canonical form – $\square$, Π, $\mathbf{Li}$, λ, ε or $;;$ –, the term t' can be reduced to a term that either has the same head, or is some $?$. We call these *catch-up lemmas*, as they enable the less precise term to "catch up" on the more precise one, whose head is already known. Their aim is to ensure that casts appearing in a less precise term never block reduction, as they can always be reduced away.

The catch-up lemmas are established in a descending fashion: first, on the universe – Lemma 10.12 –, then on types – Lemma 10.13 –, and finally on terms, namely on λ-abstractions – Lemma 10.14, and inductive constructors – Lemma 10.15. Each time, the previously proven catch-up lemmas are used to reduce types in casts appearing in the less precise term – apart from Lemma 10.12, where an induction hypothesis is used instead.

Lemma 10.12. Universe catch-up

> Under the hypothesis that $\Gamma_{.1} \sqsubseteq_\alpha \Gamma_{.2}$, if $\Gamma \vdash \square_i \sqsubseteq_\to T'$ and $\Gamma_{.2} \vdash T' \triangleright_\square \square_j$, then either $T' \to_{\mathrm{h}}^\star ?_{\square_j}$ with $i < j$, or $T' \to_{\mathrm{h}}^\star \square_i$.

Proof.

It is enough to show the property for $\sqsubseteq_\alpha$, in which case we prove the judgement by induction on T'.

We know that the judgement $\Gamma \vdash \square_i \sqsubseteq_\to T'$ must have been obtained by either Rule Univ-Diag or Rule Unk, followed by a sequence of Rule Cast-R. Thus, T' is a sequence of casts around either $\square_i$ or some $?_S$, and all types appearing in the casts are less (definitionally) precise than $\square_{i+1}$, the inferred type for $\square_i$. By induction hypothesis, they must all reduce to either some $?$ or $\square_j$. In all cases, we can show that they must reduce away. □

Lemma 10.13. Types catchup

> Under the hypothesis that $\Gamma_{.1} \sqsubseteq_\alpha \Gamma_{.2}$, we have the following:
>
> - if $\Gamma \vdash ?_{\square_i} \sqsubseteq_\alpha T'$ and $\Gamma_{.2} \vdash T' \triangleright_\square \square_j$, then $T' \to_{\mathrm{h}}^\star ?_{\square_j}$ and $i \leq j$;
> - if $\Gamma \vdash \Pi\, x{:}\, A.\, B \sqsubseteq_\alpha T'$, $\Gamma_{.1} \vdash \Pi\, x{:}\, A.B \triangleright \square_i$ and $\Gamma_{.2} \vdash T' \triangleright_\square \square_j$, then either $T' \to_{\mathrm{h}}^\star ?_{\square_j}$ and $i \leq j$, or $T' \to_{\mathrm{h}}^\star \Pi\, x{:}\, A'.B'$ for some A' and B' such that $\Gamma \vdash \Pi\, x{:}\, A.\, B \sqsubseteq_\alpha \Pi\, x{:}\, A'.B'$;
> - if $\Gamma \vdash \mathbf{Li}(A) \sqsubseteq_\alpha T'$, $\Gamma_{.1} \vdash \mathbf{Li}(A) \triangleright \square_i$ and $\Gamma_{.2} \vdash T' \triangleright_\square \square_j$, then either $T' \to_{\mathrm{h}}^\star ?_{\square_j}$ and $i \leq j$, or $T' \to_{\mathrm{h}}^\star \mathbf{Li}(A')$ for some a' such that $\Gamma \vdash \mathbf{Li}(A) \sqsubseteq_\alpha \mathbf{Li}(A')$.

Proof.

The idea of the proof is very similar to that of Lemma 10.12: decompose T' into a series of cast, and check that all those casts reduce. To do so, we need the previous lemma to know that the types appearing in the casts have a weak-head normal form of the right kind – either $?_\square$ or $\square$.

□

Lemma 10.14. λ-abstraction catch-up

> If $\Gamma \vdash \lambda\, x\colon A.t \sqsubseteq_\alpha s'$, where t is not an error, $\Gamma.{}_1 \vdash \lambda\, x\colon A.\, t \triangleright \Pi\, x\colon A.B$ and $\Gamma.{}_2 \vdash s' \triangleright_\Pi \Pi\, x\colon A'.B'$, then $s' \to_{\mathrm{h}}^\star \lambda\, x\colon A'.t'$ with $\Gamma \vdash \lambda\, x\colon A.\, t \sqsubseteq_\alpha \lambda\, x\colon A'.t'$.
>
> This holds in CastCIC$^{\mathcal{G}}$, CastCIC$^{\uparrow}$, and for terms without ? in CastCIC$^{\mathcal{N}}$.

Proof.

Again, the idea is the same. However, there is a twist here, because the lemma does *not* hold in CastCIC$^{\mathcal{N}}$ in whole generality. Indeed, there a cast through ? might error too eagerly, meaning that the whole term s' errors while t is not an error.

□

This Lemma 10.14 deserves a more extensive discussion, because it is the critical point where the difference between the three variants of CastCIC manifests, as it does not hold in full generality for CastCIC$^{\mathcal{N}}$. Indeed, the fact that $i \leq \mathrm{c}_\Pi(\mathrm{s}_\Pi(i,j))$ and $j \leq \mathrm{c}_\Pi(\mathrm{s}_\Pi(i,j))$ appears crucially in the proof to ensure that casting from a Π-type into ? and back does not reduce to an error, given the restrictions on types in Cast-R. This is the manifestation in the reduction of the embedding-projection property [NA18].

[NA18]: New et al. (2018), *Graduality from Embedding-Projection Pairs*

In CastCIC$^{\mathcal{N}}$, Lemma 10.14 still holds only if one restricts to terms without ?, where such casts never happen. This is important with regard to conservativity, as elaboration produces terms with casts but without ?, and Lemma 10.14 ensures that precision is still a simulation for these, even in CastCIC$^{\mathcal{N}}$.

The following term t_i illustrates these differences:

$$t_i \coloneqq \langle \mathrm{N} \to \mathrm{N} \Leftarrow ?_{\square_i} \Leftarrow \mathrm{N} \to \mathrm{N} \rangle\, \lambda\, x\colon \mathrm{N}.\mathrm{S}(x)$$

Such a term appears naturally whenever a loss of precision happens on a function, for instance when elaborating a term such as

$$((\lambda\, x\colon \mathrm{N}.\mathrm{S}(x)) :: ?)\, 0$$

This term t_i always reduces to

$$\langle \mathrm{N} \to \mathrm{N} \Leftarrow ?_{\square_i} \Leftarrow \mathrm{germ}_i\Pi \Leftarrow \mathrm{N} \to \mathrm{N} \rangle\, \lambda\, x\colon \mathrm{N}.\mathrm{S}(x)$$

and at this point the difference kicks in: if $\mathrm{germ}_i\Pi$ is $\mathrm{err}_{?_{\square_i}}$ – *i.e.* if $\mathrm{c}_\Pi(i) < 0$ –, then the whole term reduces to $\mathrm{err}_{\mathrm{N}\to\mathrm{N}}$. Otherwise, further reductions finally give

$$\lambda\, x\colon \mathrm{N}.\mathrm{S}\,(\langle \mathrm{N} \Leftarrow \mathrm{N} \Leftarrow \mathrm{N} \rangle\, x)$$

Although the body is blocked by the variable x, applying the function to 0 would reduce to 1 as expected. Let us compare what happens in the three systems.

In all of them, if $i \geq 1$, we have $\vdash \lambda\, x\colon \mathrm{N}.\mathrm{S}(x) \sqsubseteq_\alpha t_i$, via repeated uses of Rule Cast-R, since $\cdot \vdash \mathrm{N} \to \mathrm{N} \rightsquigarrow \mathrm{N} \to \mathrm{N} \triangleright \mathrm{s}_\Pi(0,0)$, and $\mathrm{s}_\Pi(0,0) \leq 1 \leq i$. Moreover, also $0 \leq i - 1 \leq \mathrm{c}_\Pi(i)$ and so the reduction is errorless. Thus, Lemma 10.14 holds in all three systems when $i \geq 1$.

The difference appears in the specific case where $i = 0$. In CastCIC$^{\mathcal{G}}$ and CastCIC$^{\mathcal{N}}$, we still have $\vdash \lambda\, x{:}\,\mathbf{N}.\mathrm{S}(x) \sqsubseteq_\alpha t_0$, since $\mathrm{s}_\Pi(0,0) = 0 \leq i$. In the former, $\mathrm{c}_\Pi(0) = 0$ so t_0 reduces safely and Lemma 10.14 holds. In the latter, however, $\mathrm{c}_\Pi(0) = -1$, and so t_0 errors even if it is less precise than an errorless term – Lemma 10.14 does not hold in that case. Finally, in CastCIC$^{\uparrow}$, t_0 errors since again $\mathrm{c}_\Pi(0) = -1$. However, because $s\, \mathrm{s}_\Pi(0,0) = 1$, t_0 is not less precise than $\lambda\, x{:}\,\mathbf{N}.\mathrm{S}(x)$ thanks to the typing restriction in Cast-R, so this error does not contradict Lemma 10.14.

Lemma 10.15. Constructors and inductive unknown catch-up

If $\Gamma \vdash \varepsilon_A \sqsubseteq_\alpha s'$, $\Gamma_{\cdot 1} \vdash \varepsilon_A \triangleright \mathbf{Li}(A)$ and $\Gamma_{\cdot 2} \vdash s' \triangleright_{\mathbf{Li}} \mathbf{Li}(A')$, then either $s' \to_{\mathrm{h}}^\star ?_{\mathbf{Li}(A')}$, or $s' \to_{\mathrm{h}}^\star \varepsilon_{A'}$ with $\Gamma \vdash A \sqsubseteq_\alpha A'$.

Similarly, if $\Gamma \vdash a ;;_A l \sqsubseteq_\alpha s'$, $\Gamma_{\cdot 1} \vdash a ;;_A l \triangleright \mathbf{Li}(A)$ and $\Gamma_{\cdot 2} \vdash s' \triangleright_{\mathbf{Li}} \mathbf{Li}(A')$, then either $s' \to_{\mathrm{h}}^\star ?_{\mathbf{Li}(A')}$, or $s' \to_{\mathrm{h}}^\star a' ;;_{A'} l'$ with $\Gamma \vdash A \sqsubseteq_\alpha A'$, $\Gamma \vdash a \sqsubseteq_\alpha a'$ and $\Gamma \vdash l \sqsubseteq_\alpha l'$.

Finally, if $\Gamma \vdash ?_{\mathbf{Li}(A)} \sqsubseteq_\alpha s'$, $\Gamma_{\cdot 1} \vdash ?_{\mathbf{Li}(A)} \triangleright \mathbf{Li}(A)$ and $\Gamma_{\cdot 2} \vdash s' \triangleright_{\mathbf{Li}} \mathbf{Li}(A')$, then $s' \to_{\mathrm{h}}^\star ?_{\mathbf{Li}(A')}$ with $\Gamma \vdash A \sqsubseteq_\to A'$.

Note that for Lemma 10.15, we need to deal with unknown terms specifically, which is not necessary for Lemma 10.14 because the unknown term in a Π-type reduces to a λ-abstraction.

10.3.3. Simulation

We finally come to the main property of this section, the advertised simulation property. It needs to be stated – and proven – mutually for structural and definitional precision.

Theorem 10.16. Precision is a *simulation* for reduction

Suppose we have that $\Gamma_{\cdot 1} \sqsubseteq_\to \Gamma_{\cdot 2}$, $\Gamma_{\cdot 1} \vdash t \triangleright T$, $\Gamma_{\cdot 2} \vdash u \triangleright U$ and $t \to^\star t'$. Then

- if $\Gamma \vdash t \sqsubseteq_\alpha u$, there exists u' such that $u \to^\star u'$ and $\Gamma \vdash t' \sqsubseteq_\alpha u'$;
- if $\Gamma \vdash t \sqsubseteq_\to u$ then $\Gamma \vdash t' \sqsubseteq_\to u$.

This holds in CastCIC$^{\mathcal{G}}$, CastCIC$^{\uparrow}$ and for terms without $?$ in CastCIC$^{\mathcal{N}}$.

Moreover, if $t \to^\star t'$ is replaced by $t \to_{\mathrm{h}}^\star t'$ in the hypotheses, then $u \to^\star u'$ can be replaced by $u \to_{\mathrm{h}}^\star u'$ in the conclusion.

Proof.

The case of definitional precision holds by confluence of reduction. For the case of structural precision, the hardest point is of course that of top-level, where we use Lemmas 10.14 and 10.15, to show that a similar reduction can also happen in t', once the destructed term has properly caught up.

We must also take care when handling the premises of precision where typing is involved. In particular, subject reduction is needed to relate the types inferred after reduction to the type inferred before, and the mutual induction hypothesis on $\sqsubseteq_\to$ is used to conclude that the premises holding on t still hold on t'. Finally, the restriction to terms without $?$

in CastCIC$^{\mathcal{N}}$ similar to Lemma 10.14 appears again when treating Up-Down, where having $c_\Pi\,(s_\Pi\,(i,i)) = i$ is required.

Finally, since the catch-up can be done using only weak-head reduction, a weak-head reduction step can always be simulated by weak-head reductions.

□

From this theorem, we get as direct corollaries the following properties, that are required to handle reduction – Corollary 10.17 – and consistency – Corollary 10.18 – in elaboration. Again, those corollaries hold in GCIC$^{\mathcal{G}}$, GCIC$^{\uparrow}$ and for terms in GCIC$^{\mathcal{N}}$ containing no $?$.

Corollary 10.17. Monotonicity of reduction to type constructor

Let Γ, T and T' be such that $\Gamma.1 \vdash T \rhd_\square \square_i$, $\Gamma.2 \vdash T' \rhd_\square \square_j$, and $\Gamma \vdash T \sqsubseteq_\alpha T'$. Then

- if $T \to^\star ?_{\square_i}$ then $T' \to^\star ?_{\square_j}$ with $i \leq j$;
- if $T \to^\star \square_{i-1}$ then either $T' \to^\star ?_{\square_j}$ with $i \leq j$, or $T' \to^\star \square_{i-1}$;
- if $T \to^\star \Pi\, x{:}\, A.\, B$, then either $T' \to^\star ?_{\square_j}$ with $i \leq j$, or $T' \to^\star \Pi\, x{:}\, A'.\, B'$ and $\Gamma \vdash \Pi\, x{:}\, A.B \sqsubseteq_\alpha \Pi\, x{:}\, A'.B'$;
- if $T \to^\star \mathbf{Li}\,(A)$ then either $T' \to^\star ?_{\square_j}$ with $i \leq j$, or $T' \to^\star \mathbf{Li}\,(A')$ and $\Gamma \vdash \mathbf{Li}\,(A) \sqsubseteq_\alpha \mathbf{Li}\,(A')$.

Moreover, the same hold by replacing $\to^\star$ with $\to^\star_{\mathrm{h}}$ everywhere.

Proof.

It suffices to simulate the reductions of T by using Theorem 10.16, and then use Lemmas 10.12 and 10.13 to conclude. □

Corollary 10.18. Monotonicity of consistency

If $\Gamma \vdash T \sqsubseteq_\alpha T'$, $\Gamma \vdash S \sqsubseteq_\alpha S'$ and $T \sim S$, then $T' \sim S'$.

Proof.

By definition of $\sim$, we get some U and V such that $T \to^\star U$ and $S \to^\star V$, and $U \sim_\alpha V$. By Theorem 10.16, we can simulate these reductions to get some U' and V' such that $T' \to^\star U'$ and $S' \to^\star V'$, and also $\Gamma.1 \vdash U \sqsubseteq_\alpha U'$ and $\Gamma.1 \vdash V \sqsubseteq_\alpha V'$. It remains to show that α-consistency is monotone with respect to structural precision $\sqsubseteq_\alpha$, which is direct by induction. □

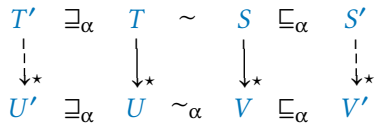

Figure 10.11. The proof of Corollary 10.18, as a diagram

10.4. Properties of GCIC

We now finally have enough technical tools to prove most of the properties of GCIC. We state those theorems in an empty context in this section to make them more readable, but they are of course corollaries of similar statements including contexts, proven by mutual induction. The complete statements and proofs can be found in Lennon-Bertrand et al. [Len+22].

[Len+22]: Lennon-Bertrand et al. (2022), *Gradualizing the Calculus of Inductive Constructions*

10.4.1. Conservativity with respect to CIC

Elaboration systematically inserts casts during checking, thus even static terms are not elaborated to themselves. Therefore, we use a (partial) erasure function eras to relate terms of CastCIC to terms of CIC by erasing all casts. We also introduce the notion of erasability, characterizing terms that contain only "harmless" casts, such that in particular the elaboration of a static term is always erasable.

Definition 10.19. Equi-precision

Two terms s and t are *equi-precise* in a context Γ, denoted $\Gamma \vdash s \sqsupseteq\sqsubseteq_\alpha t$ if both $\Gamma \vdash s \sqsubseteq_\alpha t$ and $\Gamma \vdash t \sqsubseteq_\alpha s$.

Definition 10.20. Erasure, erasability

Erasure eras is a partial function from the syntax of CastCIC to the syntax of CIC, which is undefined on $?$ and err, is such that $\mathrm{eras}(\langle B \Leftarrow A \rangle\, t) = \mathrm{eras}(t)$, and is a congruence for all other term constructors.

Given a context Γ, we say that a term t well-typed in $\Gamma_{\cdot 1}$ is *erasable* if $\mathrm{eras}(t)$ is defined, well-typed in $\Gamma_{\cdot 2}$, and equi-precise with t in Γ. Similarly, a context Γ is called *erasable* if it is pointwise erasable. When Γ is erasable, we say that a term t is erasable in Γ to mean that it is erasable in $\Gamma \mid \mathrm{eras}(\Gamma)$.

Armed with these definitions, we can state and prove conservativity. It holds in all three systems, typeability being of course taken in the corresponding variant of CIC: full CIC for $\mathrm{GCIC}^{\mathcal{G}}$ and $\mathrm{GCIC}^{\mathcal{N}}$, and $\mathrm{CIC}^{\uparrow}$ for $\mathrm{GCIC}^{\uparrow}$.

Theorem 10.21. *Conservativity*

Let t be a static term – *i.e.* a term of CIC.

If $\cdot \vdash t \triangleright T$ for some type T, then there exists t' and T' such that $\cdot \vdash t \rightsquigarrow t' \triangleright T'$, and moreover $\mathrm{eras}(t) = t$ and $\mathrm{eras}(T') = T$.

Conversely, if $\cdot \vdash t \rightsquigarrow t' \triangleright T$ for some t' and T, then $\cdot \vdash t \triangleright \mathrm{eras}(T)$.

Proof.

Because t is static, its typing derivation in GCIC can only use rules that have a counterpart in CIC, and conversely all rules of CIC have a counterpart in GCIC. The only difference is about the reduction/conversion side conditions, which are used on elaborated types in GCIC, rather than their non-elaborated counterparts in CIC.

Thus, the main difficulty is to ensure that the extra casts inserted by elaboration do not alter reduction. This is why we maintain the property that all terms t considered in CastCIC are erasable, and more precisely that any static term t that elaborates to some t' is such that $\mathrm{eras}(t) = t$. Indeed, from the simulation property of structural precision (Theorem 10.16), we obtain that an erasable term t has the same reduction behaviour as its erasure, *i.e.* if $t \to^\star s$ then $\mathrm{eras}(t) \to^\star s'$ for some s' such that $s' = \mathrm{eras}(s)$, and conversely if $\mathrm{eras}(t) \to^\star s'$ then $t \to^\star s$ for some s such that $s' = \mathrm{eras}(s)$. Using that property, we prove

$$\frac{}{x \sqsubseteq^{\mathrm{G}}_{\alpha} x} \qquad \frac{}{\square_i \sqsubseteq^{\mathrm{G}}_{\alpha} \square_i} \qquad \frac{A \sqsubseteq^{\mathrm{G}}_{\alpha} A' \qquad B \sqsubseteq^{\mathrm{G}}_{\alpha} B'}{\Pi\, x{:}\,A.B \sqsubseteq^{\mathrm{G}}_{\alpha} \Pi\, x{:}\,A'.B'} \qquad \frac{A \sqsubseteq^{\mathrm{G}}_{\alpha} A' \qquad t \sqsubseteq^{\mathrm{G}}_{\alpha} t'}{\lambda\, x{:}\,A.t \sqsubseteq^{\mathrm{G}}_{\alpha} \lambda\, x{:}\,A.t} \qquad \frac{t \sqsubseteq^{\mathrm{G}}_{\alpha} t' \qquad u \sqsubseteq^{\mathrm{G}}_{\alpha} u'}{t\ u \sqsubseteq^{\mathrm{G}}_{\alpha} t'\ u'}$$

$$\frac{A \sqsubseteq^{\mathrm{G}}_{\alpha} A'}{\mathbf{Li}\,(A) \sqsubseteq^{\mathrm{G}}_{\alpha} \mathbf{Li}\,(A')} \qquad \frac{A \sqsubseteq^{\mathrm{G}}_{\alpha} A'}{\varepsilon_A \sqsubseteq^{\mathrm{G}}_{\alpha} \varepsilon_{A'}} \qquad \frac{A \sqsubseteq^{\mathrm{G}}_{\alpha} A' \qquad a \sqsubseteq^{\mathrm{G}}_{\alpha} a' \qquad l \sqsubseteq^{\mathrm{G}}_{\alpha} l'}{a ;;_A l \sqsubseteq^{\mathrm{G}}_{\alpha} a' ;;_{A'} l'}$$

$$\frac{s \sqsubseteq^{\mathrm{G}}_{\alpha} s' \qquad P \sqsubseteq^{\mathrm{G}}_{\alpha} P' \qquad b_\varepsilon \sqsubseteq^{\mathrm{G}}_{\alpha} b'_\varepsilon \qquad b_{;;} \sqsubseteq^{\mathrm{G}}_{\alpha} b'_{;;}}{\mathrm{ind}_{\mathbf{Li}}\,(s; z.P; b_\varepsilon, y_1.y_2.p_{y_2}.b_{;;}) \sqsubseteq^{\mathrm{G}}_{\alpha} \mathrm{ind}_{\mathbf{Li}}\,(s'; z.P'; b'_\varepsilon, y_1.y_2.p_{y_2}.b'_{;;})}$$

$$\frac{}{t \sqsubseteq^{\mathrm{G}}_{\alpha} ?_i}$$

Figure 10.12. Syntactic precision for GCIC

that constrained inference on an erasable term of CastCIC behave the same as its erasure. Similarly, consistency of erasable terms of CastCIC is equivalent to conversion of the erased terms. □

10.4.2. Elaboration Graduality

Next, we turn to elaboration graduality, the equivalent of the static gradual guarantee (SGG) of Siek et al. [Sie+15] in our setting. We state it with respect to a notion of precision for terms in GCIC, *syntactic precision* $\sqsubseteq^{\mathrm{G}}_{\alpha}$, defined in Figure 10.12. Syntactic precision is the usual and expected source-level notion of precision in gradual languages: it is generated by a single non-trivial rule $t \sqsubseteq^{\mathrm{G}}_{\alpha} ?_i$, and congruence rules for all term formers.

[Sie+15]: Siek et al. (2015), *Refined Criteria for Gradual Typing*

In contrast with the simply-typed setting, the presence of multiple unknown types $?_i$, one for each universe level i, requires an additional hypothesis relating elaboration and precision judgements.

Definition 10.22. Universe adequacy

We say that two judgements $\tilde{t} \sqsubseteq^{\mathrm{G}}_{\alpha} ?_i$ and $\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright T$ are *universe adequate* if the universe level j given by the well-formation judgement $\Gamma \vdash T \triangleright_{\square} \square_j$ induced by correction of the elaboration satisfies $i = j$. More generally, $\tilde{t} \sqsubseteq^{\mathrm{G}}_{\alpha} \tilde{s}$ and $\Gamma \vdash \tilde{t} \rightsquigarrow t \triangleright T$ are *universe adequate* if for any sub-term $\tilde{t}_0$ of $\tilde{t}$ inducing judgements $\tilde{t}_0 \sqsubseteq^{\mathrm{G}}_{\alpha} ?_i$ and $\Gamma_0 \vdash \tilde{t}_0 \rightsquigarrow t_0 \triangleright T$, those are universe adequate in the previous sense.

Note that this extraneous technical assumption on universe levels should be painless in a practical system using typical ambiguity, since universe levels are very seldom given explicitly. In such a case, the elaboration would insert fresh universe levels at each $?$, which would automatically ensure universe adequacy.

Theorem 10.23. *Elaboration Graduality* / Static Gradual Guarantee

In $\mathrm{GCIC}^{\mathcal{G}}$ and $\mathrm{GCIC}^{\uparrow}$, if $\tilde{t} \sqsubseteq^{\mathrm{G}}_{\alpha} \tilde{s}$ and $\cdot \vdash \tilde{t} \rightsquigarrow t \triangleright T$ by derivations that

> are universe adequate, then $\cdot \vdash \tilde{s} \rightsquigarrow s \triangleright S$ for some s and S such that $\cdot \vdash t \sqsubseteq_\alpha s$ and $\cdot \vdash T \sqsubseteq_\alpha S$.

Proof.

The proof is by induction on the elaboration derivation for $\tilde{t}$.

All cases for inference consist in a straightforward combination of the hypotheses, with the universe adequacy hypothesis used in the case where $\tilde{s}$ is $?_i$, in order to relate the inferred types.

Here again the technical difficulties arise in the rules involving computation. This is where Corollary 10.17 is useful, proving that the less precise type obtained by induction can simulate the reduction of the more precise one. Thus, either the same rule can still be used, or one has to trade Rule Inf-Univ, Inf-Prod or Inf-List respectively for Rule Inf-Univ?, Inf-Prod? or Inf-List? in case the less precise type is some $?_\square$ and the more precise type is not.

Similarly, Corollary 10.18 proves that in the checking rule the less precise types are still consistent.

Note that, again, because Corollary 10.17 still holds when restricted to weak-head reduction, elaboration graduality also holds when fixing a weak-head strategy for elaboration. □

10.4.3. Dynamic Gradual Guarantee

[Sie+15]: Siek et al. (2015), *Refined Criteria for Gradual Typing*

Following Siek et al. [Sie+15], using the fact that structural precision is a simulation (Theorem 10.16), we can prove the DGG for CastCIC$^{\mathcal{G}}$ and CastCIC$^{\uparrow}$ – stated using the notion of observational refinement $\sqsubseteq^{\text{ob}}$ from Definition 9.4.

Theorem 10.24. Dynamic Gradual Guarantee for CastCIC$^{\mathcal{G}}$ and CastCIC$^{\uparrow}$

> Suppose that $\Gamma \vdash t \triangleright A$ and $\Gamma \vdash u \triangleright A$. If moreover $\Gamma \mid \Gamma \vdash t \sqsubseteq_\alpha u$, then $t \sqsubseteq^{\text{ob}} u$.

Proof.

Let $\mathcal{C} : (\Gamma \vdash A) \Rightarrow (\vdash \mathbf{B})$ closing over all free variables. By the diagonal rules of structural precision, we have $\Gamma \mid \Gamma \vdash \mathcal{C}[t] \sqsubseteq_\alpha \mathcal{C}[u]$. By safety (Theorem 10.4), $\mathcal{C}[t]$ either reduces to tt, ff, $?_{\mathbf{B}}$, $\mathrm{err}_{\mathbf{B}}$ or diverges, and similarly for $\mathcal{C}[u]$. If $\mathcal{C}[t]$ diverges or reduces to $\mathrm{err}_{\mathbf{B}}$, we are done. If it reduces to either tt, ff or $?_{\mathbf{B}}$, then by the catch-up Lemma 10.15, $\mathcal{C}[u]$ either reduces to the same value, or to $?_{\mathbf{B}}$. In particular, it cannot diverge or reduce to an error. □

Note that the counter-example to Lemma 10.14 given in Section 10.3 provides a counter-example to this theorem as well for CastCIC$^{\mathcal{N}}$, by choosing the context $\mathrm{ind}_{\mathbf{N}}(\bullet\ 0; z.\mathbf{B}; \mathrm{tt}, \mathrm{tt})$, because in that context the function $\lambda\, x{:}\,\mathbf{N}.\ \mathrm{S}(x)$ reduces to tt while the less precise cast function reduces to $\mathrm{err}_{\mathbf{B}}$.

11. Beyond CastCIC: Models, Indices and Pure Reasoning

Chapter 10 establishes quite a few properties of GCIC and CastCIC, culminating with elaboration graduality. This is, however, still far from satisfactory. First, it is missing proofs of normalization for the two variants which are supposed to satisfy it – CastCIC$^{\mathcal{N}}$ and CastCIC$^{\uparrow}$ –, and of graduality – for CastCIC$^{\mathcal{G}}$ and CastCIC$^{\uparrow}$. Next, it only treats CIC$_{_}$, *i.e.* it does not handle indexed inductive types. But these are crucial in order to really exploit dependency: for instance, most of our introductory examples were based on the vector type, an indexed inductive type outside the scope of Chapter 10.

In this chapter we go over these issues and possible solutions: Section 11.1 describes model constructions to establish both normalization and graduality; Section 11.2 describes the issue with indexed inductive types, and gives possible solutions in case the indices are "nice enough", which covers vectors; finally, Section 11.3 gives a much more ambitious solution to handle indices, giving a proper treatment of the equality type – the stereotypical pathologic inductive type –, and much more.

As these have no direct relation to bidirectional typing *per se*, we do not dwell on the technical details in this chapter. The interested reader can consult either Lennon-Bertrand et al. [Len+22] – Section 11.1 corresponds roughly to Section 6 there, and Section 11.2 to Sections 6 and 8.3 –, or Maillard et al. [Mai+22] – corresponding to Section 11.3.

[Len+22]: Lennon-Bertrand et al. (2022), *Gradualizing the Calculus of Inductive Constructions*

[Mai+22]: Maillard et al. (2022), *A Reasonably Gradual Type Theory*

11.1. Realizing CastCIC

11.1.1. The discrete model

To inform the design and justify the reduction rules provided for CastCIC, we build a syntactic model[1] of CastCIC by translation to CIC augmented with induction-recursion [Mar96; DS03; GMF15].

From a type theoretical point of view, what makes CastCIC peculiar are the possibility of having *exceptions* – both "pessimistic" (err) and "optimistic" (?) –, and the necessity to do *intensional type analysis* in order to resolve casts. For the former, we build upon the work of Pédrot and Tabareau [PT18] on the exceptional type theory ExTT. For the latter, we reuse the technique of Boulier, Pédrot, and Tabareau [BPT17] to equip the universe with an elimination principle $typerec$[2], which requires induction-recursion to be implemented.

We call this syntactic model of CastCIC the *discrete model*. It captures the intuition that the unknown type is inhabited by "tagged values", *e.g.* a term together with its type. In other words, the unknown type $?_{\square}$ behaves as a dependent sum $\Sigma\, A : \square.\ A$. Projecting out of it is realized through type analysis using $typerec$, and may fail – raising an error in the ExTT sense.

Note that we provide a particular interpretation of the unknown term in the universe, which is legitimized by an observation made by Pédrot and

1: *Syntactic models* [BPT17; Bou18], are a kind of models of type theory defined directly by induction on the raw syntax, in a way akin to program translation or compilation. This allows for simple models, that moreover can be used to capture fine-grained properties that only make sense on that raw syntax, typically those that need to separate between convertible terms.

[BPT17]: Boulier et al. (2017), *The next 700 syntactical models of type theory*

[Bou18]: Boulier (2018), *Extending Type Theory with Syntactical Models*

[Mar96]: Martin-Löf (1996), *On the Meanings of the Logical Constants and the Justifications of the Logical Laws*

[DS03]: Dybjer et al. (2003), *Induction-recursion and initial algebras*

[GMF15]: Ghani et al. (2015), *Positive Inductive-Recursive Definitions*

[PT18]: Pédrot et al. (2018), *Failure is Not an Option An Exceptional Type Theory*

2: Corresponding to a form of ad-hoc polymorphism.

Tabareau [PT18]: ExTT does not constrain in any way the definition of exceptions in the universe. This is crucial to combine ExTT with a universe equipped with $typerec$.

The key point to prove normalization is that reduction is preserved, in the sense that a reduction step in the source theory CastCIC is mapped to at least one step in the target. Thus, the target being normalizing, so is CastCIC.

Theorem 11.1. Normalization for CastCIC

> Both CastCIC$^{\mathcal{N}}$ and CastCIC$^{\uparrow}$ have the normalization property (Property 3.14).

An important corollary of this property, in combination with safety (Theorem 10.4), is a weak form of logical consistency, characterizing the possible inhabitants of the empty type:

Theorem 11.2. Weak logical consistency

> Suppose t is a closed inhabitant of the empty type $\bot$ in CastCIC$^{\mathcal{N}}$ or CastCIC$^{\uparrow}$, *e.g.* $\vdash t \triangleleft \bot$. Then t must reduce to either $\mathrm{err}_{\bot}$ or $?_{\bot}$.

11.1.2. The monotone model

The simplicity of the discrete model comes at the price of an inherent inability to characterize which casts are guaranteed to succeed, *i.e.* a graduality theorem. To overcome this limitation, and prove graduality of CastCIC$^{\uparrow}$, we can build a more elaborate *monotone model*, inducing a precision relation that is well-behaved with respect to conversion.

In this model, each type A comes equipped with an order structure $\sqsubseteq^{A}$ – a reflexive, transitive, antisymmetric and proof-irrelevant relation – modelling precision between terms. In particular, the exceptions err_A and $?_A$ correspond respectively to the smallest and greatest element of A for this order. Saying that this interpretation of types as posets is a model is equivalent to saying that each term and type former is enforced to be monotone, providing a strong form of graduality. This implies in particular that such a model cannot be defined for CastCIC$^{\mathcal{N}}$, as this type theory lacks graduality.

The precision order of the monotone model can be reflected back to CastCIC, giving rise to the *propositional precision* judgment $\Gamma \vdash t \ {}_T\sqsubseteq_U \ u$, where T and U are the respective types of t and u. Type dependency naturally demands such a notion of inhomogeneous precision, rather than a simpler notion relating only terms of the same type.

This precision relation bears a similar relationship to definitional precision $\sqsubseteq_{\to}$ as propositional equality to conversion/definitional equality in CIC. Propositional precision can be used to prove precision statements inside the target type theory, for instance we can show by case analysis on $b : \mathbf{B}$ that

$$b : \mathbf{B} \vdash \mathrm{ind}_{\mathbf{B}}(b; x.\Box; A, A) \ {}_{\Box}\sqsubseteq_{\Box} \ A$$

a judgment that does not hold for definitional precision. In particular, propositional precision is invariant by conversion: if $t \cong t'$, $u \cong u'$ and $\Gamma \vdash t \ {}_T\sqsubseteq_U \ u$ then $\Gamma \vdash t' \ {}_T\sqsubseteq_U \ u'$. But this means that propositional precision is too coarse to capture properties such as the simulation property (Theorem 10.16) and its corollaries (Corollaries 10.17 and 10.18), because these distinguish convertible terms.

Still, we can relate the two notions, as follows:

Theorem 11.3. Compatibility of structural and propositional precision

> If $\Gamma \vdash t \triangleright T$, $\Gamma \vdash u \triangleright U$ and $\Gamma \mid \Gamma \vdash t \sqsubseteq_\alpha u$, then $\Gamma \vdash t \ {}_T\sqsubseteq_U \ u$.
>
> Conversely, if $\cdot \vdash v_1 \ {}_{\mathbf{B}}\sqsubseteq_{\mathbf{B}} \ v_2$ for normal forms v_1, v_2, then $\cdot \vdash v_1 \sqsubseteq_\alpha v_2$.

Again, this is similar to the relation between conversion and propositional equality: the former always implies the latter, and one can come back from the second to the first in a constrained enough setting – here, on closed booleans.

Finally, the main property satisfied by propositional precision is graduality:

Theorem 11.4. Graduality

> Propositional precision satisfies the Dynamic Gradual Guarantee: if $\Gamma \vdash t \triangleright T$, $\Gamma \vdash t' \triangleright T$ and $\Gamma \vdash t \ {}_T\sqsubseteq_T \ t'$ hold, then $t \sqsubseteq^{\mathrm{ob}} t'$.
>
> Casts form embedding-projection pairs. That is, if $\Gamma \vdash t \triangleright T$ and $\Gamma \vdash u \triangleright U$, and moreover $\Gamma \vdash T \ {}_\square\sqsubseteq_\square \ U$, then the following three properties are equivalent:
>
> $$\Gamma \vdash \langle U \Leftarrow T \rangle \, t \ {}_U\sqsubseteq_U \ u \Leftrightarrow \Gamma \vdash t \ {}_T\sqsubseteq_U \ u \Leftrightarrow \Gamma \vdash t \ {}_T\sqsubseteq_T \ \langle T \Leftarrow U \rangle \, u$$
>
> And furthermore, $\Gamma \vdash \langle T \Leftarrow U \rangle \, \langle U \Leftarrow T \rangle \, t \ {}_T\sqsubseteq_T \ t$ – this is the retraction property.

11.2. The issue with indices: gradual vectors and equalities

11.2.1. The issue with propositional equality

For the sake of exposing the problem, suppose that we can define the equality type $a =_A a'$ in CastCIC, while still satisfying canonicity, conservativity and graduality. This means that for an equality $t = u$ involving closed terms t and u of CIC, there should only be three possible canonical forms: $\mathrm{refl}_{A,t}$ whenever t and u are convertible terms, as well as err and $?$.

Just under these assumptions, we can show that there exist two functions that are pointwise equal in CIC, but are no longer equivalent in CastCIC. Consider the two functions $\mathrm{id}_{\mathbb{N}}$ and $\mathrm{add0}$ defined respectively as $\mathrm{id}_{\mathbb{N}} := \lambda\, x\colon \mathbb{N}.\, n$ and $\mathrm{add0} := \lambda\, x\colon \mathbb{N}.\, x + 0$. In CIC, these functions are not convertible, but they are pointwise equal, and observationally equivalent. However,

they would not be observationally equivalent in GCIC under our assumptions. To see why, consider the following term:

$$\mathrm{test} := \lambda\, f : \mathbf{N} \to \mathbf{N}.\mathrm{ind}_{=}\,(y.z.\mathbf{B}; \langle \mathrm{id}_{\mathbf{N}} = f \Leftarrow ?_{\square} \Leftarrow \mathrm{id}_{\mathbf{N}} = \mathrm{id}_{\mathbf{N}} \rangle\, \mathrm{refl}; \mathrm{tt})$$

We have $\mathrm{test}\,\mathrm{id}_{\mathbf{N}} \to^{\star} \mathrm{tt}$ because, by graduality, the upcast-downcast in the scrutinee must succeed, *i.e.*

$$\langle \mathrm{id}_{\mathbf{N}} = \mathrm{id}_{\mathbf{N}} \Leftarrow ?_{\square} \Leftarrow \mathrm{id}_{\mathbf{N}} = \mathrm{id}_{\mathbf{N}} \rangle\, \mathrm{refl} \to^{\star} \mathrm{refl}$$

However, since add0 is *not* convertible to $\mathrm{id}_{\mathbf{N}}$,

$$\langle \mathrm{id}_{\mathbf{N}} = \mathrm{add0} \Leftarrow ?_{\square} \Leftarrow \mathrm{id}_{\mathbf{N}} = \mathrm{id}_{\mathbf{N}} \rangle\, \mathrm{refl}$$

cannot possibly reduce to refl, and thus would need to reduce either to err or ?; and so does test add0. This means that the theory would be very intensional, by being able to distinguish between any two functions that are not convertible, even if they are pointwise equal.

More generally, the issue is the following: given a constructor c of an inductive type I, we need to decide what to do when confronted with some $\langle I(b'_1, \dots, b'_n) \Leftarrow I(b_1, \dots, b_n) \rangle\, c(a_1, \dots, a_m)$. If I does not have indices, as in the case of lists, we know that c can always be used to inhabit $I(b'_1, \dots, b'_n)$, if given arguments of the right types. If I has indices, however, this might not be possible due to typing constraints, as in the example of refl. But we still need to provide an inhabitant of that type as redex for the cast! If we resort to the wildcards ? or err, then we expose a very intensional behaviour such as the one above. However, in the setting of a generic inductive type – such as that of the equality –, deciding whether it is inhabited by a "valid", non-wildcard, term in a given non-empty context is undecidable, so we cannot hope to always decide whether the cast should fail or return such a valid term.

11.2.2. Solutions for vectors

3: And quite a lot of those used in the context of dependently typed programming.

Thankfully, not all indexed inductive types are as thorny as equality. Indeed, in examples such as **Ve**,[3] solutions are possible that avoid the dead-end identified for equality, by carefully using the structure of indices. These rely on two well-known alternatives to indexed inductive types for capturing properties intrinsically: type-level eliminators, and "forded" inductive types.

Type-level eliminators. Instead of an inductive type, **Ve** can be defined as a recursive function on its index, at the type level, effectively representing lists as nested pairs:

$$\mathrm{Ve}^{\mu} := \lambda(A : \square)(n : \mathbf{N}).\mathrm{ind}_{\mathbf{N}}(n; x.\square; \top, y.p_y.A \times p_y)$$

corresponding to the Coq definition

[BMM04]: Brady et al. (2004), *Inductive Families Need Not Store Their Indices*
[Gil+19]: Gilbert et al. (2019), *Definitional Proof-Irrelevance without K*

```
Fixpoint FVect (A : Type) (n : ℕ) : Type :=
  match n with 0 ⇒ unit | S n ⇒ A * FVec A n end.
```

Type-level eliminators can be used as soon as the indices are *concretely forceable* [BMM04]. Intuitively, concretely forceable indices are those that can be matched upon (like n in the example of **Ve**). Gilbert et al. [Gil+19] give a general translation of this kind to build mock-up inductive types inside a sort of definitionally irrelevant propositions.

This presentation coincides with the indexed inductive one on standard, non-exceptional terms, but quickly becomes very imprecise in presence of unknown indices.

Forded inductive type. Instead of using an indexed inductive type, one can use a parametrized inductive type, with *explicit* equalities as arguments to constructors.[4] For instance, vectors can be defined in this style as follows:

```
Inductive Vectf (A : Type) (n : ℕ) : Type :=
| nilf : eq_nat 0 n → Vectf A n
| consf : A → forall m : ℕ, eq_nat (S m) n
    → Vectf A m → Vectf A n.
```

Here, `eq_nat` is the type of *decidable* equality proofs between natural numbers, expressing the constraints on n – *e.g.* $n = 0$ – but avoiding the use of the unavailable propositional equalities –, which can be defined like this:

```
Fixpoint eq_nat (m n : ℕ) : Type :=
match m, n with
  | 0, 0 ⇒ True
  | S m, S n ⇒ eq_nat m n
  | _, _ ⇒ False
end.
```

This presentation is more accurate than the previous one when dealing with unknown indices, but is too permissive with invalid index assertions. It fails very late when such invalid assertions are made, meaning that error-reporting is bad.

Direct support. In contrast to the two previously-exposed encodings that both have serious shortcomings, extending CastCIC with direct support for indexed inductive types can provide a much more satisfactory solution. The idea is to reason about indices directly in the reduction of casts.

To do so, we first add two new canonical forms, corresponding to the casts of ε and $;;$ to $\mathrm{Ve}(A, ?_{\mathbf{N}})$: namely, $\varepsilon^{?}_{A}$ and $a\ ;;^{?}_{A,n}\ v$.

We then extend reduction to account for casts on vectors in canonical forms. Figure 11.1a presents these rules when the argument of the cast is a non-empty vector. Rule V-cons-? propagates the cast on the arguments, but using the newly introduced $;;^{?}$. This effectively loses precision in the type information, but keeps it all recorded in the term, so that it can be used in case of a downcast. This is exactly what Rule V-cons?-s does. Rule V-cons-s applies when both source and target indices are successors, and propagates the cast of the arguments, just like in the case of lists. Finally, as expected, Rule V-cons-0 raises an error when the indices do not match. Similarly, Rule V-cons?-0 also raises an error when trying to create a vector of length 0 from one with an unknown index, but whose underlying vector is non-empty.

4: This technique has been coined "forded" by McBride [McB99, Section 3.5], as an allusion to Henry Ford' quote "*Any customer can have a car painted any color that he wants, so long as it is black.*"

[McB99]: McBride (1999), *Dependently typed functional programs and their proofs*

$$\text{V-cons-?:}\ \langle \mathrm{Ve}(B, ?_{\mathbb{N}}) \Leftarrow \mathrm{Ve}(A, \mathrm{S}(n))\rangle\ (a\ ;;_{A,k} v) \rightharpoonup (\langle B \Leftarrow A\rangle\, a)\ ;;^{?}_{B,n}\ (\langle \mathrm{Ve}(B,n) \Leftarrow \mathrm{Ve}(A,k)\rangle\, v)$$

$$\text{V-cons?-s:}\ \langle \mathrm{Ve}(B, \mathrm{S}(n)) \Leftarrow \mathrm{Ve}(A, ?_{\mathbb{N}})\rangle\ (a\ ;;^{?}_{A,k} v) \rightharpoonup (\langle B \Leftarrow A\rangle\, a)\ ;;_{B,n}\ (\langle \mathrm{Ve}(B,n) \Leftarrow \mathrm{Ve}(A,k)\rangle\, v)$$

$$\text{V-cons-s:}\ \langle \mathrm{Ve}(B, \mathrm{S}(m)) \Leftarrow \mathrm{Ve}(A, \mathrm{S}(n))\rangle\ (a\ ;;_{A,k} v) \rightharpoonup (\langle B \Leftarrow A\rangle\, a)\ ;;_{B,m}\ (\langle \mathrm{Ve}(B,m) \Leftarrow \mathrm{Ve}(A,k)\rangle\, v)$$

$$\text{V-cons-0:}\ \langle \mathrm{Ve}(B, 0) \Leftarrow \mathrm{Ve}(A, \mathrm{S}(n))\rangle\ (a\ ;;_{A,k} v) \rightharpoonup \mathrm{err}_{\mathrm{Ve}(B,0)}$$

$$\text{V-cons?-0:}\ \langle \mathrm{Ve}(B, 0) \Leftarrow \mathrm{Ve}(A, ?_{\mathbb{N}})\rangle\ (a\ ;;^{?}_{A,k} v) \rightharpoonup \mathrm{err}_{\mathrm{Ve}(B,0)}$$

Figure 11.1a. Casts between gradual vector types (excerpt)

Figure 11.1b. Eliminator for the gradual vector type (excerpt)

$$\text{V-rect-nil?:}\ \mathrm{ind}_{\mathbf{Ve}}\,(\varepsilon^{?}_{A}; y.z.P; b_{\varepsilon}, y_1.y_2.y_3.p_{y_3}.b_{;;}) \rightharpoonup \langle P ?_{\mathbb{N}} \Leftarrow P 0\rangle\, b_{\varepsilon}$$

For the eliminator, there are two new computation rules, one for each new constructor. We give the one for the case of $\varepsilon^{?}$, this is Rule V-rect-nil?. These rules apply the eliminator to the underlying non-exceptional constructor, and then cast the result to $P ?_{\mathbb{N}}$. Intuitively, they transfer the cast on vectors to a cast on the reduct.

Note that all of these rules crucially use the fact that it is possible to discriminate between 0, $\mathrm{S}(n)$ and $?_{\mathbb{N}}$, which is a specificity of the vector type and explains why this solution is not possible for *e.g.* the equality.

This "definitive" presentation is justified by a modification of the models described in Section 11.1, and gives the satisfactory behaviour described in the examples of Sections 9.0.1 and 9.0.4: it preserves as much computational content as possible, while failing early when invalid assumptions are used.

11.3. A Reasonably Gradual Type Theory

In the context of a gradual proof assistant based on CIC, the normalizing and conservative variant GCIC$^{\mathcal{N}}$ is the most appealing, as it ensures decidability of typing, (weak) canonicity, and supports all existing developments and libraries by virtue of being a conservative extension of CIC. Unfortunately, the universe shift introduced in CastCIC$^{\mathcal{N}}$ during reduction means that some terms break graduality. For instance, while the term

$$\mathrm{nArrow} \coloneqq \lambda\, n\colon \mathbb{N}.\ \mathrm{ind}_{\mathbb{N}}(n; x.\square_0; \mathbb{N}, y.p_y.\mathbb{N} \to p_y)$$

or, in Coq,

```
Fixpoint nArrow (n : ℕ) : Type :=
match n with
  | 0 ⇒ ℕ
  | S m ⇒ ℕ → nArrow n
end.
```

is well-typed in GCIC$^{\mathcal{N}}$, the type $\Pi\, n\colon \mathbb{N}.\, \mathrm{nArrow}\, n$ does not satisfy the embedding-projection property with respect to any unknown type $?_{\square_i}$, because the appropriate universe level is not known *a priori*. However, apart

from the fact that GCIC$^{\mathcal{N}}$ does not satisfy graduality globally, little is known about its gradual properties as its metatheory in this regard has not been developed. In particular, there is no clear characterization of a class of terms for which graduality holds.

A refined stratification of precision However, by refining the stratification of precision we can develop a full account of graduality for an extension of CastCIC$^{\mathcal{N}}$, called *GRIP*. The key idea is that $?_{\square_i}$ should be the least precise type among all types at level i and below, *except* for dependent function types at level i – which are however still less precise than $?_{\square_{i+1}}$.

We can precisely characterize problematic terms as those that are not *self-precise* – *i.e.* more precise than themselves. For function types, self-precision means monotonicity with respect to precision. A recursive large elimination as in nArrow is not monotone because there is no fixed level i for which $\mathrm{nArrow}\ n \sqsubseteq ?_{\square_i}$, given $n \sqsubseteq ?_{\mathrm{N}}$.

On the contrary, we can prove that the dynamic gradual guarantee holds in GRIP for *any* self-precise context, and that casts between types related by precision induce embedding-projection pairs between self-precise terms. Therefore, this shift in perspective in the interpretation of the unknown type and the associated notion of precision yields a gradual theory that conservatively extends CIC, is normalizing, and satisfies graduality for a large and well-defined class of terms.

Internalizing precision, reasonably While we could study graduality for GRIP externally, we observe that we can exploit the expressiveness of the type-theoretic setting to internalize precision and its associated reasoning. In particular, this makes it possible to state and prove, within the theory itself, results about (self-)precision and graduality for specific terms. In a way, this is the natural next step following the definition of propositional precision in Section 11.1.

Introducing *internal precision* in a gradual type theory however requires us to address two main obstacles. First, when adding exceptions to CIC [PT18], the theory becomes inconsistent as a logic, because it is possible to inhabit any type A by raising an exception err_A. In the gradual setting, there is also the alternative of using the unknown term $?_A$ to inhabit any type A. If we want to support valid internal reasoning about precision and graduality, we need to avoid these degenerate proofs and provide a logically consistent theory. Second, the gradual type theory needs to satisfy extensionality principles in order to support the notion of precision as error approximation [NA18]. Embracing extensionality principles in an intensional type theory such as CIC is a challenge in itself.

[PT18]: Pédrot et al. (2018), *Failure is Not an Option An Exceptional Type Theory*

[NA18]: New et al. (2018), *Graduality from Embedding-Projection Pairs*

We can address both issues by combining recent advances in type theory: the reasonably exceptional type theory *RETT* [Péd+19] and the observational type theory *TTObs* [PT22]. First, RETT supports consistent reasoning about exceptional terms. It features a layer of possibly exceptional terms, and a separate layer of pure terms in which raising an exception is prohibited. This way, the consistency of the logical layer is guaranteed, while allowing non-trivial interaction with the exceptional layer. Technically, the two layers are defined using two distinct universe hierarchies. Second, based on the seminal work on Observational Type Theory [AMS07],

[Péd+19]: Pédrot et al. (2019), *A Reasonably Exceptional Type Theory*

[PT22]: Pujet et al. (2022), *Observational Equality: Now for Good*

[AMS07]: Altenkirch et al. (2007), *Observational Equality, Now!*

[Gil+19]: Gilbert et al. (2019), *Definitional Proof-Irrelevance without K*

TTObs provides a setoidal equality in a specific universe SProp of definitionally proof-irrelevant propositions. This universe of strict propositions, introduced by [Gil+19] and supported in recent versions of Coq and Agda, makes it possible to define an extensional notion of equality, while trivializing the so-called higher coherence hell by imposing that any two proofs of a given equality are *convertible*.

A major insight of GRIP is to realize that we can actually merge the logical universe of RETT used to reason about exceptional terms with the universe SProp of proof-irrelevant propositions in order to define an internal notion of precision that is extensional and whose proofs cannot be trivialized with exceptional terms.

Applications of internal precision In addition to supporting reasoning about the graduality of terms in a theory that is not globally gradual, internal precision makes it possible to support gradual subset types, in which a type can be refined by a proposition expressed using precision. Moreover, in the literature, exception handling is never considered when proving graduality because this mechanism inherently allows terms that do not behave monotonically with respect to precision. Internal precision enables us to support exception handling in the impure layer of the type theory, and to consistently reason about the graduality (or not) of exception-handling terms.

12. Perspectives

I hope that this thesis gives compelling arguments for the adoption of bidirectional typing, but there is more.

Part 'Bidirectional Calculus of Inductive Constructions' shows that one can use the valuable bidirectional structure, without having to leave their favourite declarative system behind. Indeed, it can most likely be shown equivalent to a bidirectional one – given it is one's favourite system, it does surely satisfy the good properties needed for that.

In Part 'A Certified Kernel for Coq, in Coq', we enter the real world, and see how this plays out on a complex type system: bidirectionalism gives good guidelines to analyse typing rules, and provides a precise specification to prove the implementation correct, while allowing separation of concerns. If this is enough to catch bugs in Coq, it should prove useful in finding those in other kernels, too. Yet, MetaCoq is much more than bidirectional typing: its certified kernel opens up a new era for proof assistants, with a previously unreached trust level.

Finally, Part 'Bidirectional Elaboration for Gradual Typing' exemplifies how the bidirectional structure can be useful when simply designing a type system, even without a single implementation in sight. But gradual typing can hopefully be more than a mere example. As it enables the transition of programmers from the soft realms of dynamic typing to the discipline of static typing, so it could open the door of dependently typed programming to more than a fraction of fanatic enthusiasts.

Still, as most thesis, this one opens up at least as many questions as it answers, in all its three broad directions.

12.1. Bidirectional Typing for Dependent Types

The formal study of bidirectional typing in the setting of dependent types still begs for more investigations. While I hope the present work gives a robust answer in the setting of Curry-style syntax, where every term infers a type, the case of Church-style syntax is quite different. In the case of normal forms, the proof ideas presented in this thesis should be easily adapted. But if we wish to go beyond normal forms, we must consider the use of annotations in terms, as is done in *e.g.* McBride [McB22], Gratzer, Sterling, and Birkedal [GSB19] or Dunfield and Krishnaswami [DK21]. However, due to the dependently-typed setting, we have to investigate how these annotations play out with conversion and/or reduction. To the best of my knowledge, only McBride has taken that question up, but does not arrive – yet – at a definitive solution, so there is matter left for further research.

[McB22]: McBride (2022), *Types Who Say Ni*

[GSB19]: Gratzer et al. (2019), *Implementing a Modal Dependent Type Theory*

[DK21]: Dunfield et al. (2021), *Bidirectional Typing*

Another thread to pull is the relation with Generalized Type Systems [BHL20; BP22]. Here, as in McBride's discipline, we find well-formation invariants to be preserved, and carefully structured rules that should respect them. Recasting the bidirectional concepts in such a setting could allow for a better

[BHL20]: Bauer et al. (2020), *A general definition of dependent type theories*

[BP22]: Bauer et al. (2022), *An extensible equality checking algorithm for dependent type theories*

understanding both of the ideas at work in bidirectional typing, yielding a proper formal account of McBride's discipline together with a general proof that it ensures good properties of the system, and of the well-formation conditions already explored in Bauer, Haselwarter, and Lumsdaine [BHL20] on judgment boundaries.

Since Generalized Type Systems put conversion and typing on the same footing, it also seems natural to question how we can marry conversion and bidirectionalism. Here again there are ingredients in the air: Abel, Öhman, and Vezzosi [AÖV17] show a notion of conversion geared towards proving decidability of typing, but which is clearly bidirectional, and could serve as a basis to give a general notion of bidirectional conversion. This subject is only scratched in Chapter 6, but I believe that the ideas presented there can be scaled to a system such as PCUIC, and be an interesting building block in order to specify extensionality rules as used in Coq's kernel.

[AÖV17]: Abel et al. (2017), *Decidability of Conversion for Type Theory in Type Theory*

12.2. MetaCoq's Future

MetaCoq is a mature project, and has reached the stage where the formalization can really serve as a tool to move Coq forward.

We have already evoked in Chapter 5 the question of the representation of pattern-matching. This is a relatively minor question, but more complex ones – *e.g.* the integration of a sort SProp of strict propositions, or subject reduction for co-inductive types – can now be investigated in MetaCoq, providing a valuable guidance to their implementation in Coq.

However, MetaCoq is still quite some distance away from type-checking realistic developments in Coq, as it lacks some important features present in the latter's kernel. Barring template polymorphism,[1] there are two main lacking elements that are to be integrated if we wish to really reach the project's goal.

1: A restricted form of universe polymorphism, which the latter should hopefully be able to replace.

The first are extensionality conversion rules: definitional proof irrelevance, and η laws for functions and records. The η conversion laws are basic features, present in virtually any modern proof assistant. However, in the precise context of MetaCoq, they pose subtle questions.[2] Broadly, giving a specification of such η laws is easy in the setting of typed conversion, but much trickier in that of untyped conversion. However, the whole structure of MetaCoq is built around that untyped notion of conversion, and could not be so easily adapted to a typed conversion. This makes the integration of η laws challenging. The case of strict propositions is less well-known, being much more recent, but poses similar challenges. A possible solution to solve these issues would be to move the whole development over to typed conversion, using the ideas introduced in Chapter 6.

2: For an overview of these, see Lennon-Bertrand [Len22].

[Len22]: Lennon-Bertrand (2022), *À bas l'η – Coq's troublesome η-conversion*

The second lacking feature are modules and functors. While these are less pervasive than η laws, they are still present in a number of developments. Here again the difficulty is not simply to show that an implementation is faithful to a given semantic, but to precisely pin down said semantic. This is tricky in the case of modules, which have interactions with global environments, contrarily to records – their first-class counterpart. This unclear semantic is probably one of the reason modules are not used more, and so

putting them on a stabler ground might also give users more confidence to use them.

A last important investigation to make MetaCoq closer to the real kernel is that of guard conditions. The impossibility to prove full normalization of PCUIC does not mean that we should not completely abandon this question. We can at least implement a guard condition, and show that it fulfils the conditions we abstractly ask for in the current development. More ambitious, the complex guard condition implemented in Coq was designed [Gim95] in order to allow a translation back to eliminators. This gives a much stronger validity criterion for the guard, but would not be an easy project. But as for modules, reaching that goal would make the guard much more trustworthy than it currently is. Moreover, it could open the door to extending it, with the formalization as a safeguard as to the validity of those extensions.

[Gim95]: Giménez (1995), *Codifying guarded definitions with recursive schemes*

Beyond these missing but rather necessary pieces, MetaCoq should hopefully offer tools for broader investigations around Coq's core: formalization of tactics, of syntax transformation and generation... Some have already started to appear [FK19; LUF20], but hopefully more are to come!

[FK19]: Forster et al. (2019), *A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus*
[LUF20]: Liesnikov et al. (2020), *Generating induction principles and subterm relations for inductive types using MetaCoq*

12.3. Gradual CIC

As for the last part of this thesis, if the aim goal of gradual typing is to answer the needs of developers, we should get closer to those. I believe that GRIP gives at least a good starting point to experiment with, so the main missing piece now is an implementation. Such an implementation of course is no small feat: integrating a new feature to CIC is never easy, even more so one of this scale. Moreover, it raises subtle questions. For instance, while almost all reduction rules of dependent type theories are parametric over the universe levels, reduction in CastCIC crucially depends on those. In a setting where these universe levels are not mere integers, what becomes of those? How do we handle a non-total order between universe levels?

Appendix

A. Names for Type Systems

MLTT and CIC

In the field of dependently types, I think we can safely delineate two main schools, with different histories and cultures. The first goes back to Martin-Löf – in particular Martin-Löf [Mar72] –, and is strongly linked to the Agda proof assistant. The second is related to the proof assistant Coq, in the filiation of Coquand and Huet – since Coquand and Huet [CH88]. The umbrella name "MLTT", for Martin-Löf Type Theory is the one usually used for systems in the first school, while ones in the second tend to use "CIC" – Calculus of Inductive Constructions –, or variants thereof.

[Mar72]: Martin-Löf (1972), *An intuitionistic theory of types*

[CH88]: Coquand et al. (1988), *The calculus of constructions*

This separation is of course not a strict one, and researchers from both schools interact, exchange theoretical and implementation ideas, and move forward together. But still, this cultural difference is not anecdotal, as seemingly small differences between the approaches on both sides lead to wildly different behaviours between the systems, so that some techniques that are very successful on one side can prove unusable on the other.

I tried to probe the community of proof assistants[1] as to what they consider the more important differences between the two schools, which led to quite different answers, although this is very approximate: Agda has a general scheme for inductive types (including cubical ones in the cubical library) while many articles on CIC only consider a few example inductive types – as was the case in parts of this thesis –, etc. So this should be read as "this tradition is more prone to taking that approach". The results are summarized in Figure A.1.

1: Using a question on the proof assistant Stack Exchange.

	Universes	Inductive Types	Pattern-matching	Philosophy	Conversion
MLTT	Predicative hierarchy	0, 1, W and Id	Top-level clauses	Constructivism	Typed
CIC	Impredicative Prop	General scheme	First-class terms	None/Formalism	Untyped

Figure A.1. General characteristics of MLTT and CIC

Why "CIC"?

The one feature which came out maybe as the more prominent in the distinction between MLTT and CIC is the presence of an impredicative sort of propositions, which immensely augments the logical power of the theory, and makes it much harder to prove normalization. Despite the exclusion of propositions by default, I still chose to use the name CIC in this thesis, for multiple reasons.

First, regarding all other columns in the table, the system fits more in the bottom line than the top one. In particular, the general spirit of studying conversion using tools from rewriting theory which appears as a repeated pattern throughout the thesis is incompatible – or, at the very least, must be heavily amended – with a typed conversion. Moreover, apart from Part 'Bidirectional Elaboration for Gradual Typing', the absence of treatment

of Prop on the paper presentation was done mostly due to simplification concerns than to theoretical limitations, as the formalization of PCUIC as a whole illustrates. This also applies to Chapter 10, even though the models presented in Chapter 11 do not scale to Prop, meaning that the target of Chapter 10 would then be on a precarious foundation.

But more importantly, in the bidirectional approach, there is again a clear cultural difference between Agda/MLTT and Coq/CIC. The former have used the bidirectional ideas for a long time in order to allow for a lightweight syntax using Curry-style abstractions, at the cost of losing completeness of typing on non-normal forms.[2] The latter insist on keeping enough annotations in the kernel syntax by using Church-style abstractions to let every term infer, and use a mechanism of implicit arguments during elaboration to lighten the weight of for users. This means that the completeness theorem as stated in Theorem 4.3 does *not* hold in any of the standard presentations of MLTT, while it does to CIC's, as this thesis shows.

2: This is a deliberate trade-off, at least in the case of Agda [Nor07, p. 19].

[Nor07]: Norell (2007), *Towards a practical programming language based on dependent type theory*

Bibliography

Here are the references in citation order.

[Har20] Kevin Hartnett. 'Building the Mathematical Library of the Future'. In: *Quanta Magazine* (Oct. 1, 2020). (Visited on 04/20/2022) (cited on pages 1, 13).

[Fre79] Gottlob Frege. *Begriffsschrift: Eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens*. Halle a.d.S.: Louis Nebert, 1879 (cited on pages 3, 4, 15, 16, 25).

[Ded72] Richard Dedekind. *Stetigkeit und Undirrationale Zahlen*. Braunschweig: F. Vieweg und Sohn, 1872 (cited on pages 3, 15).

[Can72] Georg Cantor. 'Ueber die Ausdehnung eines Satzes aus der Theorie der trigonometrischen Reihen'. In: *Mathematische Annalen* 5 (1872), pp. 123–132 (cited on pages 3, 15).

[Pea89] Giuseppe Peano. *Arithmetices principia: Nova methodo exposita*. Torino: Fratres Bocca, 1889 (cited on pages 3, 15).

[Can83] Georg Cantor. *Grundlagen einer allgemeinen Mannigfaltigkeitslehre. Ein mathematisch-philosophischer Versuch in der Lehre des Unendlichen*. Leibnitz: Teubner, 1883 (cited on pages 3, 15).

[Fre03] Gottlob Frege. *Grundgesetze der Arithmetik*. Vol. 2. Pohle: Jena, 1903 (cited on pages 3, 15).

[WR13] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1913 (cited on pages 4, 15, 16).

[Zer08] Ernst Zermelo. 'Untersuchungen über die Grundlagen der Mengenlehre I'. In: *Mathematische Annalen* 65 (1908), pp. 261–281 (cited on pages 4, 15).

[Zer04] E. Zermelo. 'Beweis, daß jede Menge wohlgeordnet werden kann'. In: *Mathematische Annalen* 59.4 (1904), pp. 514–516. DOI: 10.1007/BF01445300 (cited on pages 4, 15).

[Göd31] Kurt Gödel. 'Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme. I'. In: *Monatshefte für Mathematik und Physik* 37 (1931), pp. 173–198 (cited on pages 4, 15, 16).

[dBru70] Nicolaas Govert de Bruijn. 'The mathematical language AUTOMATH, its usage, and some of its extensions'. In: *Symposium on automatic demonstration*. Springer. 1970, pp. 29–61 (cited on pages 5, 17).

[Rud92] Piotr Rudnicki. 'An overview of the Mizar project'. In: *Proceedings of the 1992 Workshop on Types for Proofs and Programs*. 1992, pp. 311–330 (cited on pages 5, 17).

[Voe10] Vladimir Voevodsky. 'Univalent foundations project'. In: *NSF grant application* (2010) (cited on pages 5, 17).

[Hal12] Thomas Hales. *Dense Sphere Packings: A Blueprint for Formal Proofs*. London Mathematical Society Lecture Note Series. Cambridge University Press, 2012 (cited on pages 5, 17).

[Sch21] Peter Scholze. *Half a year of the Liquid Tensor Experiment: Amazing developments*. Blog post. 2021. URL: https://xenaproject.wordpress.com/2021/06/05/half-a-year-of-the-liquid-tensor-experiment-amazing-developments/ (visited on 05/18/2022) (cited on pages 5, 17).

[Del00] David Delahaye. 'A Tactic Language for the System Coq'. In: *Proceedings of Logic for Programming and Automated Reasoning (LPAR)*. Vol. 1955. LNCS/LNAI. Jan. 2000, pp. 85–95 (cited on pages 5, 17).

[Bla+16] Jasmin C. Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. 'Hammering towards QED'. In: *Journal of Formalized Reasoning* 9.1 (Jan. 2016), pp. 101–148. DOI: 10.6092/issn.1972-5787/4593 (cited on pages 5, 17).

[Eki+17] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. 'SMTCoq: A plug-in for integrating SMT solvers into Coq'. In: *Computer Aided Verification - 29th International Conference*. Heidelberg, Germany, July 2017 (cited on pages 5, 17).

[LW22] Robert Y. Lewis and Minchao Wu. 'A Bi-Directional Extensible Interface Between Lean and Mathematica'. In: *Journal of Automated Reasoning* (2022). DOI: 10.1007/s10817-021-09611-1 (cited on pages 6, 17).

[MMS19] Assia Mahboubi, Guillaume Melquiond, and Thomas Sibut-Pinote. 'Formally Verified Approximations of Definite Integrals'. In: *Journal of Automated Reasoning* 62.2 (Feb. 2019), pp. 281–300. DOI: 10.1007/s10817-018-9463-7 (cited on pages 6, 17).

[Käs+17] Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. 'Closing the Gap – The Formally Verified Optimizing Compiler CompCert'. In: *SSS'17: Safety-critical Systems Symposium 2017*. Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium. Bristol, United Kingdom: CreateSpace, Feb. 2017, pp. 163–180 (cited on pages 6, 17).

[Bha+17] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. 'Everest: Towards a Verified, Drop-in Replacement of HTTPS'. In: *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Ed. by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Vol. 71. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 1:1–1:12. DOI: 10.4230/LIPIcs.SNAPL.2017.1 (cited on pages 6, 17).

[Imm18] Fabian Immler. 'A Verified ODE Solver and the Lorenz Attractor'. In: *Journal of Automated Reasoning* 61.1 (2018), pp. 73–111. DOI: 10.1007/s10817-017-9448-y (cited on pages 6, 17).

[Coq22a] The Coq Development Team. *The Coq Proof Assistant*. Version 8.15. Jan. 2022. DOI: 10.5281/zenodo.5846982 (cited on pages 6, 17).

[Mil78] Robin Milner. 'A theory of type polymorphism in programming'. In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4 (cited on pages 6, 18, 44).

[How80] William Howard. 'The Formulae-as-Types Notion of Construction'. In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* 44 (1980). Original paper manuscript from 1969, pp. 479–490 (cited on pages 6, 18).

[CFC58] Haskell Curry, Robert Feys, and William Craig. *Combinatory Logic*. Vol. 1. North Holland Publishing Company, 1958 (cited on pages 6, 18).

[BG01] Henk Barendregt and Herman Geuvers. 'Proof-Assistants Using Dependent Type Systems'. In: *Handbook of Automated Reasoning*. Ed. by Alan Robinson and Andrei Voronkov. Handbook of Automated Reasoning. Amsterdam: North-Holland, 2001, pp. 1149–1238. DOI: 10.1016/B978-044450813-3/50020-5 (cited on pages 8, 19).

[Hue89] Gérard Huet. 'The Constructive Engine'. In: *A Perspective in Theoretical Computer Science*. 1989, pp. 38–69. DOI: 10.1142/9789814368452_0004. eprint: https://www.worldscientific.com/doi/pdf/10.1142/9789814368452_0004 (cited on pages 9, 20, 49).

[Sie+15] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 'Refined Criteria for Gradual Typing'. In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Ed. by Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 274–293. DOI: 10.4230/LIPIcs.SNAPL.2015.274 (cited on pages 9, 20, 107, 111, 113, 116–118, 123, 126, 132, 141, 149, 150).

[Len+22] Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. 'Gradualizing the Calculus of Inductive Constructions'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2022). DOI: 10.1145/3495528 (cited on pages 11, 22, 107, 108, 129, 141, 147, 151).

[Mai+22] Kenji Maillard, Meven Lennon-Bertrand, Nicolas Tabareau, and Éric Tanter. *A Reasonably Gradual Type Theory*. Mar. 2022. Hal: hal-03596652 (cited on pages 11, 22, 108, 151).

[Len21] Meven Lennon-Bertrand. 'Complete Bidirectional Typing for the Calculus of Inductive Constructions'. In: *12th International Conference on Interactive Theorem Proving (ITP 2021)*. Ed. by Liron Cohen and Cezary Kaliszyk. Vol. 193. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. DOI: 10.4230/LIPIcs.ITP.2021.24 (cited on pages 11, 22).

[SLF22] Matthieu Sozeau, Meven Lennon-Bertrand, and Yannick Forster. 'The Curious Case of Case: Correct & Efficient Representation of Case Analysis in Coq and MetaCoq'. Talk. 1st Workshop on the Implementation of Type Systems, 2022 (cited on pages 11, 22, 64).

[Len22] Meven Lennon-Bertrand. 'À bas l'η – Coq's troublesome η-conversion'. Talk. 1st Workshop on the Implementation of Type Systems, 2022 (cited on pages 11, 22, 65, 160).

[BHL20] Andrej Bauer, Philipp G. Haselwarter, and Peter LeFanu Lumsdaine. *A general definition of dependent type theories*. 2020. arXiv: 2009.05539 (cited on pages 24, 52, 69, 159, 160).

[Ayd+05] Brian Aydemir, Aaron Bohannon, Matthew Fairbairn, Nathan Foster, Benjamin Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 'Mechanized metatheory for the masses: the POPLmark challenge'. In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2005, pp. 50–65 (cited on page 24).

[CH88] Thierry Coquand and Gérard Huet. 'The calculus of constructions'. In: *Information and Computation* 76.2 (1988), pp. 95–120. DOI: 10.1016/0890-5401(88)90005-3 (cited on pages 24, 42, 165).

[Bar92] Henk Barendregt. 'Lambda Calculi with Types'. In: *Handbook of Logic in Computer Science*. 1992 (cited on page 25).

[Nor07] Ulf Norell. 'Towards a practical programming language based on dependent type theory'. PhD thesis. Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007 (cited on pages 25, 49, 57, 58, 66, 166).

[AÖV17] 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). DOI: 10.1145/3158111 (cited on pages 25, 57, 65, 66, 73, 76, 160).

[McB22] Conor McBride. 'Types Who Say Ni'. 2022 (cited on pages 25, 58, 159).

[Pal98] Erik Palmgren. 'On universes in type theory'. eng. In: *Twenty Five Years of Constructive Type Theory*. Oxford University Press, 1998. DOI: 10.1093/oso/9780198501275.003.0012 (cited on page 25).

[Gir72] Jean-Yves Girard. 'Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur'. PhD thesis. Université Paris VII, 1972 (cited on page 25).

[Mar72] Per Martin-Löf. *An intuitionistic theory of types*. 1972 (cited on pages 25, 27, 35, 66, 165). Reprinted in Giovanni Sambin and Jan Smith. *Twenty five years of constructive type theory*. Vol. 36. Clarendon Press, 1998.

[HP91] Robert Harper and Robert Pollack. 'Type checking with universes'. In: *Theoretical Computer Science* 89.1 (1991). DOI: 10.1016/0304-3975(90)90108-T (cited on page 26).

[Bar85] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics. Revised Edition*. Studies in Logics and the Foundation of Mathematics 103. North Holland, 1985 (cited on page 27).

[Bar91] Henk Barendregt. 'An Introduction to Generalized Type Systems'. In: *Journal of Functional Programming* 1 (Apr. 1991), pp. 125–154. DOI: 10.1017/S0956796800020025 (cited on pages 27, 53, 123, 125).

[HB21] Philipp G. Haselwarter and Andrej Bauer. *Finitary type theories with and without contexts*. 2021. DOI: 10.48550/ARXIV.2112.00539 (cited on page 31).

[Tak95] M. Takahashi. 'Parallel Reductions in λ-Calculus'. In: *Information and Computation* 118.1 (1995), pp. 120–127. DOI: 10.1006/inco.1995.1057 (cited on pages 33, 92, 133).

[WF94] Andrew Wright and Matthias Felleisen. 'A Syntactic Approach to Type Soundness'. In: *Information and Computation* 115.1 (1994), pp. 38–94. DOI: 10.1006/inco.1994.1093 (cited on page 34).

[Tai67] William W Tait. 'Intensional interpretations of functionals of finite type I'. In: *The journal of symbolic logic* 32.2 (1967), pp. 198–212 (cited on page 35).

[Abe13] Andreas Abel. 'Normalization by Evaluation: Dependent Types and Impredicativity'. Habilitation thesis. Institut für Informatik, Ludwig-Maximilians-Universität München, 2013 (cited on page 35).

[Geu01] Herman Geuvers. 'Induction Is Not Derivable in Second Order Dependent Type Theory'. In: *Typed Lambda Calculi and Applications*. Ed. by Samson Abramsky. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 166–181 (cited on page 35).

[Pau93] C. Paulin-Mohring. 'Inductive Definitions in the System Coq - Rules and Properties'. In: *Proceedings of the conference Typed Lambda Calculi and Applications*. Ed. by M. Bezem and J.-F. Groote. Lecture Notes in Computer Science 664. LIP research report 92-49. 1993 (cited on page 35).

[MS84] Per Martin-Löf and Giovanni Sambin. *Intuitionistic Type Theory*. Studies in Proof Theory 1. Napoli: Bibliopolis, 1984 (cited on page 35).

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013 (cited on pages 39, 66).

[LW11] Gyesik Lee and Benjamin Werner. 'Proof-irrelevant model of CC with predicative induction and judgmental equality'. In: *Logical Methods in Computer Science* Volume 7, Issue 4 (Nov. 2011). DOI: 10.2168/LMCS-7(4:5)2011 (cited on pages 41, 65).

[Pol92] R. Pollack. 'Typechecking in Pure Type Systems'. In: *Informal Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden*. June 1992, pp. 271–288 (cited on pages 43, 44, 53, 54).

[ST14] Matthieu Sozeau and Nicolas Tabareau. 'Universe Polymorphism in Coq'. In: *Interactive Theorem Proving*. Ed. by Gerwin Klein and Ruben Gamboa. Cham: Springer International Publishing, 2014, pp. 499–514 (cited on page 44).

[Hin69] R. Hindley. 'The Principal Type-Scheme of an Object in Combinatory Logic'. In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60 (cited on page 44).

[TS18] Amin Timany and Matthieu Sozeau. 'Cumulative Inductive Types In Coq'. In: *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*. Ed. by Hélène Kirchner. Vol. 108. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 1–16. DOI: 10.4230/LIPIcs.FSCD.2018.29 (cited on page 44).

[Gim95] Eduarde Giménez. 'Codifying guarded definitions with recursive schemes'. In: *Types for Proofs and Programs*. Ed. by Peter Dybjer, Bengt Nordström, and Jan Smith. Springer Berlin Heidelberg, 1995, pp. 39–59 (cited on pages 44, 45, 161).

[CST20] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. 'Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi'. In: *FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction*. 5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020) 167. Paris, France, June 2020, 34:1–34:21. DOI: 10.4230/LIPIcs.FSCD.2020.34 (cited on page 45).

[Abe+13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 'Copatterns: programming infinite structures by observations'. In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 27–38. DOI: 10.1145/2429069.2429075 (cited on page 45).

[Jim96] Trevor Jim. 'What Are Principal Typings and What Are They Good For?' In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, pp. 42–53. DOI: 10.1145/237721.237728 (cited on page 49).

[PT00] Benjamin C. Pierce and David N. Turner. 'Local Type Inference'. In: *ACM Transactions on Programming Languages and Systems* 22.1 (Jan. 2000), pp. 1–44. DOI: 10.1145/345099.345100 (cited on page 49).

[DK21] Jana Dunfield and Neel Krishnaswami. 'Bidirectional Typing'. In: *ACM Computing Surveys* 54.5 (May 2021). DOI: 10.1145/3450952 (cited on pages 49, 52, 99, 159).

[McB18] Conor McBride. 'Basics of Bidirectionalism'. Blog post. Aug. 6, 2018. (Visited on 05/30/2022) (cited on pages 49, 51, 57, 97).

[McB19] Conor McBride. 'Check the Box!' In: *25th International Conference on Types for Proofs and Programs*. June 14, 2019 (cited on pages 49, 51, 57).

[Coq96] Thierry Coquand. 'An algorithm for type-checking dependent types'. In: *Science of Computer Programming* 26.1 (1996). DOI: 10.1016/0167-6423(95)00021-6 (cited on pages 49, 57).

[Asp+12] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. 'A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions'. In: *Logical Methods in Computer Science* Volume 8, Issue 1 (Mar. 2012). DOI: 10.2168/LMCS-8(1:18)2012 (cited on pages 49, 55, 58).

[AA11] Andreas Abel and Thorsten Altenkirch. 'A Partial Type Checking Algorithm for Type:Type'. In: *Electronic Notes in Theoretical Computer Science* 229.5 (2011). Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008), pp. 3–17. DOI: 10.1016/j.entcs.2011.02.013 (cited on pages 54, 57).

[ACD08] Andreas Abel, Thierry Coquand, and Peter Dybjer. 'Verifying a Semantic $\beta\eta$-Conversion Test for Martin-Löf Type Theory'. In: *Mathematics of Program Construction*. Ed. by Philippe Audebaud and Christine Paulin-Mohring. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 29–56 (cited on pages 54, 57).

[GSB19] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 'Implementing a Modal Dependent Type Theory'. In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341711 (cited on pages 54, 58, 159).

[Saï97] Amokrane Saïbi. 'Typing Algorithm in Type Theory with Inheritance'. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '97. doi:10.1145/263699.263742* (1997). DOI: 10.1145/263699.263742 (cited on pages 54, 55).

[Soz07] Matthieu Sozeau. 'Subset Coercions in Coq'. In: *Types for Proofs and Programs*. Ed. by Thorsten Altenkirch and Conor McBride. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 237–252 (cited on pages 54, 55).

[AC07] Andreas Abel and Thierry Coquand. 'Untyped Algorithmic Equality for Martin-Löf's Logical Framework with Surjective Pairs'. In: *Fundamenta Informaticae* 77.4 (2007). TLCA'05 special issue., pp. 345–395 (cited on page 57).

[McB16] Conor McBride. 'I Got Plenty o' Nuttin''. In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Ed. by Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella. Springer International Publishing, 2016, pp. 207–233. DOI: 10.1007/978-3-319-30936-1_12 (cited on page 57).

[Gil+19] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 'Definitional Proof-Irrelevance without K'. In: *Proceedings of the ACM on Programming Languages*. POPL'19 3.POPL (Jan. 2019), pp. 1–28. DOI: 10.1145/329031610.1145/3290316 (cited on pages 66, 154, 155, 158).

[PT22] Loïc Pujet and Nicolas Tabareau. 'Observational Equality: Now for Good'. In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498693 (cited on pages 66, 157).

[App22] Andrew W. Appel. 'Coq's Vibrant Ecosystem for Verification Engineering (Invited Talk)'. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2022. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 2–11. DOI: 10.1145/3497775.3503951 (cited on page 66).

[Wer94] Benjamin Werner. 'Une Théorie des Constructions Inductives'. Theses. Université Paris-Diderot - Paris VII, May 1994 (cited on page 73).

[Alt93] Thorsten Altenkirch. 'Constructions, Inductive Types and Strong Normalization'. PhD thesis. University of Edinburgh, 1993 (cited on page 73).

[Soz+20] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 'Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq'. In: *Proceedings of the ACM on Programming Languages* (Jan. 2020), pp. 1–28. DOI: 10.1145/3371076 (cited on pages 73, 100).

[TS17] Amin Timany and Matthieu Sozeau. *Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC)*. Research Report RR-9105. KU Leuven, Belgium ; Inria Paris, Oct. 2017, p. 32 (cited on page 86).

[Aba+91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. 'Explicit substitutions'. In: *Journal of Functional Programming* 1.4 (1991), pp. 375–416. DOI: 10.1017/S0956796800000186 (cited on page 90).

[STS15] Steven Schäfer, Tobias Tebbi, and Gert Smolka. 'Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions'. In: *Interactive Theorem Proving*. Ed. by Christian Urban and Xingyuan Zhang. Cham: Springer International Publishing, 2015, pp. 359–374 (cited on page 90).

[SM19] Matthieu Sozeau and Cyprien Mangin. 'Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq'. In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341690 (cited on page 93).

[PT17] Pierre-Marie Pédrot and Nicolas Tabareau. 'An Effectful Way to Eliminate Addiction to Dependence'. In: *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2017. DOI: 10.1109/LICS.2017.8005113 (cited on page 95).

[Win20] Théo Winterhalter. 'Formalisation and meta-theory of type theory'. PhD thesis. Université de Nantes, 2020 (cited on page 101).

[ST06] Jeremy G. Siek and Walid Taha. 'Gradual Typing for Functional Languages'. In: *In Scheme and Functional Programming Workshop*. 2006, pp. 81–92 (cited on pages 107, 110, 112, 115, 116, 127, 128, 131, 132).

[Ou+04] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 'Dynamic Typing with Dependent Types'. In: *Proceedings of the IFIP International Conference on Theoretical Computer Science*. 2004, pp. 437–450 (cited on page 107).

[WF09] Philip Wadler and Robert Bruce Findler. 'Well-Typed Programs Can't Be Blamed'. In: *Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP 2009)*. Ed. by Giuseppe Castagna. Lecture Notes in Computer Science. Springer-Verlag, 2009, pp. 1–16 (cited on pages 107, 131).

[KF10] Kenneth Knowles and Cormac Flanagan. 'Hybrid type checking'. In: *ACM Transactions on Programming Languages and Systems* 32.2 (Jan. 2010) (cited on page 107).

[TT15] Éric Tanter and Nicolas Tabareau. 'Gradual Certified Programming in Coq'. In: *Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015)*. Pittsburgh, PA, USA: ACM Press, Oct. 2015, pp. 26–40 (cited on page 107).

[LT17] Nico Lehmann and Éric Tanter. 'Gradual Refinement Types'. In: *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. Paris, France: ACM Press, Jan. 2017, pp. 775–788 (cited on page 107).

[DTT18] Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. 'Foundations of Dependent Interoperability'. In: *Journal of Functional Programming* 28 (2018) (cited on page 107).

[MT21] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, Jan. 2021 (cited on page 109).

[ETG19] Joseph Eremondi, Éric Tanter, and Ronald Garcia. 'Approximate Normalization for Gradual Dependent Types'. In: *Proceedings of the ACM on Programming Languages* 3.ICFP (Aug. 2019) (cited on pages 110, 120, 124, 128).

[GCT16] Ronald Garcia, Alison M. Clark, and Éric Tanter. 'Abstracting Gradual Typing'. In: *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL 2016)*. St Petersburg, FL, USA: ACM Press, Jan. 2016, pp. 429–442 (cited on pages 112, 115–117, 120, 128, 131).

[PT18] Pierre-Marie Pédrot and Nicolas Tabareau. 'Failure is Not an Option An Exceptional Type Theory'. In: *ESOP 2018 - 27th European Symposium on Programming*. Vol. 10801. LNCS. Thessaloniki, Greece: Springer, 2018, pp. 245–271. DOI: 10.1007/978-3-319-89884-1_9 (cited on pages 114, 129, 133, 151, 157).

[Péd+19] Pierre-Marie Pédrot, Nicolas Tabareau, Hans Jacob Fehrmann, and Éric Tanter. 'A Reasonably Exceptional Type Theory'. In: *ICFP 2019 - 24th ACM SIGPLAN International Conference on Functional Programming*. Berlin, Germany: ACM, Aug. 2019. DOI: 10.1145/3341712 (cited on pages 114, 157).

[PT20] Pierre-Marie Pédrot and Nicolas Tabareau. 'The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects'. In: *Proceedings of the ACM on Programming Languages* 4.POPL (2020). DOI: 10.1145/3371126 (cited on page 114).

[BGT16] Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 'Gradual type-and-effect systems'. In: *Journal of Functional Programming* 26 (2016). DOI: 10.1017/S0956796816000162 (cited on page 115).

[FT13] Luminous Fennell and Peter Thiemann. 'Gradual Security Typing with References'. In: *2013 IEEE 26th Computer Security Foundations Symposium*. 2013, pp. 224–239. DOI: 10.1109/CSF.2013.22 (cited on page 115).

[TGT18] Matías Toro, Ronald Garcia, and Éric Tanter. 'Type-Driven Gradual Security with References'. In: *ACM Trans. Program. Lang. Syst.* 40.4 (Dec. 2018). DOI: 10.1145/3229061 (cited on page 115).

[TF14] Peter Thiemann and Luminous Fennell. 'Gradual Typing for Annotated Type Systems'. In: *Programming Languages and Systems*. Ed. by Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 47–66 (cited on page 115).

[BMT10] Gavin Bierman, Erik Meijer, and Mads Torgersen. 'Adding Dynamic Types to C#'. In: *ECOOP 2010 – Object-Oriented Programming*. Ed. by Theo D'Hondt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 76–100 (cited on page 115).

[HTF10] David Herman, Aaron Tomb, and Cormac Flanagan. 'Space-efficient gradual typing'. In: *Higher-Order and Symbolic Computation* 23.2 (2010), pp. 167–189 (cited on page 116).

[TF08] Sam Tobin-Hochstadt and Matthias Felleisen. 'The Design and Implementation of Typed Scheme'. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: Association for Computing Machinery, 2008, pp. 395–406. DOI: 10.1145/1328438.1328486 (cited on page 116).

[SW10] Jeremy G. Siek and Philip Wadler. 'Threesomes, with and without Blame'. In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. Madrid, Spain: Association for Computing Machinery, 2010, pp. 365–376. DOI: 10.1145/1706299.1706342 (cited on pages 116, 130).

[SGT09] Jeremy Siek, Ronald Garcia, and Walid Taha. 'Exploring the Design Space of Higher-Order Casts'. In: *Programming Languages and Systems*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 17–31 (cited on page 116).

[TT20] Matías Toro and Éric Tanter. 'Abstracting gradual references'. In: *Science of Computer Programming* 197 (2020), p. 102496. DOI: 10.1016/j.scico.2020.102496 (cited on page 116).

[Bañ+21] Felipe Bañados Schwerter, Alison M. Clark, Khurram A. Jafery, and Ronald Garcia. 'Abstracting Gradual Typing Moving Forward: Precise and Space-Efficient'. In: *Proceedings of the ACM on Programming Languages* 5.POPL (Jan. 2021). DOI: 10.1145/3434342 (cited on page 116).

[NA18] Max S. New and Amal Ahmed. 'Graduality from Embedding-Projection Pairs'. In: *Proceedings of the ACM on Programming Languages* 2.ICFP (July 2018). DOI: 10.1145/3236768 (cited on pages 117, 118, 121, 130, 145, 157).

[ISI17] Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 'On Polymorphic Gradual Typing'. In: *Proceedings of the ACM on Programming Languages* 1.ICFP (Aug. 2017). DOI: 10.1145/3110284 (cited on page 117).

[Cas+19] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 'Gradual Typing: A New Perspective'. In: *Proc. ACM Program. Lang.* 3.POPL (2019). DOI: 10.1145/3290329 (cited on page 120).

[Eis16] Richard Eisenberg. *Dependent Types in Haskell: Theory and Practice*. 2016. arXiv: 1610.07978 (cited on pages 121, 126, 139).

[Bra13] Edwin Brady. 'Idris, a general-purpose dependently typed programming language: Design and implementation'. In: *Journal of Functional Programming* 23.5 (2013), pp. 552–593 (cited on pages 121, 124).

[GT20] Ronald Garcia and Éric Tanter. 'Gradual Typing as if Types Mattered'. In: *Informal Proceedings of the ACM SIGPLAN Workshop on Gradual Typing (WGT20)*. 2020 (cited on page 124).

[Ngu+19] Phúc Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 'Size-Change Termination as a Contract: Dynamically and Statically Enforcing Termination for Higher-Order Programs'. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 845–859. DOI: 10.1145/3314221.3314643 (cited on page 124).

[Coq22b] The Coq Development Team. *The Coq proof assistant reference manual*. Version 8.15. Jan. 2022. URL: https://coq.inria.fr/refman/ (visited on 04/11/2022) (cited on page 127).

[ST07] Jeremy Siek and Walid Taha. 'Gradual Typing for Objects'. In: *ECOOP 2007 – Object-Oriented Programming*. Ed. by Erik Ernst. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 2–27 (cited on page 128).

[Bou18] S. Boulier. 'Extending Type Theory with Syntactical Models'. PhD thesis. École Mines-Telecom Atlantique, 2018 (cited on pages 128, 151).

[Lev04] Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Vol. 2. Semantics Structures in Computation. Springer, 2004 (cited on page 131).

[MM94] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Universitext. Springer New York, 1994 (cited on page 131).

[FF02] Robert Bruce Findler and Matthias Felleisen. 'Contracts for Higher-Order Functions'. In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '02. Pittsburgh, PA, USA: Association for Computing Machinery, 2002, pp. 48–59. DOI: 10.1145/581478.581484 (cited on page 131).

[CS16] Matteo Cimini and Jeremy G. Siek. 'The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems'. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 443–455. DOI: 10.1145/2837614.2837632 (cited on page 137).

[BPT17] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 'The next 700 syntactical models of type theory'. In: *Certified Programs and Proofs (CPP 2017)*. Paris, France, Jan. 2017, pp. 182–194. DOI: 10.1145/3018610.3018620 (cited on page 151).

[Mar96] Per Martin-Löf. 'On the Meanings of the Logical Constants and the Justifications of the Logical Laws'. In: *Nordic Journal of Philosophical Logic* 1.1 (1996), pp. 11–60 (cited on page 151).

[DS03] Peter Dybjer and Anton Setzer. 'Induction-recursion and initial algebras'. In: *Annals of Pure and Applied Logic* 124.1-3 (2003), pp. 1–47. DOI: 10.1016/S0168-0072(02)00096-9 (cited on page 151).

[GMF15] Neil Ghani, Lorenzo Malatesta, and Fredrik Nordvall Forsberg. 'Positive Inductive-Recursive Definitions'. In: *Logical Methods in Computer Science* 11.1 (2015). DOI: 10.2168/LMCS-11(1:13)2015 (cited on page 151).

[BMM04] Edwin Brady, Conor McBride, and James McKinna. 'Inductive Families Need Not Store Their Indices'. In: *Types for Proofs and Programs*. Ed. by Stefano Berardi, Mario Coppo, and Ferruccio Damiani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 115–129 (cited on pages 154, 155).

[McB99] Conor McBride. 'Dependently typed functional programs and their proofs'. PhD thesis. University of Edinburgh, 1999 (cited on page 155).

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. 'Observational Equality, Now!' In: *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*. PLPV '07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 57–68. DOI: 10.1145/1292597.1292608 (cited on page 157).

[BP22] Andrej Bauer and Anja Petković Komel. 'An extensible equality checking algorithm for dependent type theories'. In: *Logical Methods in Computer Science* Volume 18, Issue 1 (Jan. 2022). DOI: 10.46298/lmcs-18(1:17)2022 (cited on page 159).

[FK19] Yannick Forster and Fabian Kunze. 'A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus'. In: *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Ed. by John Harrison, John O'Leary, and Andrew Tolmach. Vol. 141. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 17:1–17:19. DOI: 10.4230/LIPIcs.ITP.2019.17 (cited on page 161).

[LUF20] Bohdan Liesnikov, Marcel Ullrich, and Yannick Forster. *Generating induction principles and subterm relations for inductive types using MetaCoq*. 2020. DOI: 10.48550/ARXIV.2006.15135 (cited on page 161).

DOCTORAT BRETAGNE LOIRE / MATHSTIC

Nantes Université

Titre : Typage Bidirectionnel pour le Calcul des Constructions Inductives

Mot clés : Théorie des Types, Assistants à la Preuve, Typage Bidirectionnel, Calcul des Constructions Inductives, Coq, Typage Graduel

Résumé : Durant leurs plus de 50 ans d'existence, les assistants à la preuve se sont établis comme des outils permettant un haut niveau de fiabilité dans de nombreuses applications. Cependant, du fait de leur complexité grandissante, la solution historique de faire confiance à un petit noyau stable n'est plus suffisante pour avancer en évitant des bugs critiques. Mais les assistants à la preuve sont utilisés depuis des décennies pour certifier la correction de programmes, pourquoi pas *la leur* ? C'est l'ambition du projet MetaCoq, visant à construire le premier noyau réaliste à la correction formellement prouvée, pour l'assistant à la preuve Coq. Ne faites plus confiance au programme, seulement à sa preuve !

Cette thèse étudie la structure bidirectionnelle qui sous-tend l'algorithme de typage implémenté par le noyau de Coq, dans le contexte du Calcul des Constructions Inductives (CIC) qui fonde celui-ci. Le tout est formalisé dans le cadre de MetaCoq, et constitue un passage obligé pour atteindre l'objectif du projet, fournissant un intermédiaire entre l'implémentation et sa spécification. Enfin, le contrôle renforcé sur le calcul offert par le typage bidirectionnel est une pièce nécessaire à la conception d'une extension graduelle de CIC, qui vise à apporter au développement en Coq la flexibilité du typage dynamique et constitue la dernière partie de la thèse.

Title: Bidirectional Typing for the Calculus of Inductive Constructions

Keywords: Type Theory, Proof Assistant, Bidirectional Typing, Calculus of Inductive Constructions, Coq, Gradual Typing

Abstract: Over their more than 50 years of existence, proof assistants have established themselves as tools guaranteeing high trust levels in many applications. Yet, due to their increasing complexity, the historical solution of relying on a small, trusted kernel is not enough anymore to avoid critical bugs while moving forward. But proof assistants have been used for decades to certify program correctness, so why not *their own*? This is the ambition of the MetaCoq project, which aims at providing the first realistic kernel for a proof assistant – Coq – to be formally proven correct, in Coq itself. Don't trust the program anymore, only its proof!

This thesis studies the bidirectional structure on which the typing algorithm implemented by the kernel of Coq relies, in the context of the Calculus of Inductive Constructions on which it is founded. This is formalized as a part of MetaCoq, and is a key step to reach the project's goal, by giving an intermediate layer between the implementation and its specification. Moreover, the increased control over computation offered by bidirectional typing is a necessary piece in designing a gradual extension of CIC, which aims at bringing to development in Coq the flexibility of dynamic typing, and forms the last part of the thesis.