COLLEGE DOCTORAL PAYS DE LA LOIRE | ED MATHEMATIQUES STIC

Nantes Université

THÈSE DE DOCTORAT DE

NANTES UNIVERSITÉ

ÉCOLE DOCTORALE N° 641
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Enzo CRANCE

Meta-Programming for Proof Transfer in Dependent Type Theory

Thèse présentée et soutenue à Nantes, le 19 décembre 2023
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes (LS2N)

Rapporteurs avant soutenance :

Sandrine BLAZY — Professeur des universités, Université de Rennes, IRISA
Anders MÖRTBERG — *Associate professor*, Université de Stockholm

Composition du Jury :

Président :	Guillaume MELQUIOND	Directeur de recherche, Inria, Université Paris-Saclay, LMF
Examinateurs :	Sandrine BLAZY	Professeur des universités, Université de Rennes, IRISA
	Anders MÖRTBERG	*Associate professor*, Université de Stockholm
	Karl PALMSKOG	*Senior lecturer*, *KTH Royal Institute of Technology*, Stockholm
	Nicolas TABAREAU	Directeur de recherche, Inria, Nantes Université
	Enrico TASSI	Chargé de recherche, Inria Sophia Antipolis Méditerranée, Université Côte d'Azur
Dir. de thèse :	Assia MAHBOUBI	Directrice de recherche, Inria, LS2N, Nantes
Co-dir. de thèse :	Denis COUSINEAU	Chercheur industriel (docteur), Mitsubishi Electric R&D Centre Europe, Rennes

Invité :

Mathieu BOESPFLUG — Chercheur industriel (docteur), Modus Create

Doctoral Thesis

Meta-Programming for Proof Transfer in Dependent Type Theory

Enzo Crance

February 13, 2024

Supervised by Assia Mahboubi and Denis Cousineau

À feu Marcel Coignard, dit « Monsieur l'abbé »,
Un homme extraordinaire et un puits de culture,
qui regrettait de ne jamais avoir pu explorer les sciences exactes.

Acknowledgements

En premier lieu, je souhaite remercier mes encadrants et les chercheurs avec lesquels j'ai travaillé au long de cette thèse, sans qui ce document n'aurait jamais pu voir le jour.

Merci Assia, pour ton honnêteté intellectuelle, ta rigueur scientifique sans faille, ta patience, ta pédagogie, ta modestie, ton humanité et ton ouverture d'esprit.

Merci Denis, pour ta prise de recul permanente, tes remarques pertinentes, ton optimisme, ta sympathie et tes précieux conseils sur le plan professionnel.

Merci à vous deux d'avoir mis au point ce sujet de thèse, de m'avoir fait confiance pendant ces trois années et de toujours avoir été très disponibles.

Merci Cyril, pour ton esprit vif, tes idées innovantes et tes plans sur la comète, sources de motivation dans la recherche.

Merci Chantal, pour nos échanges productifs et cette facilité que tu as à travailler en équipe. Merci également à Louise, Valentin et Pierre pour leur travail sur notre publication commune.

Merci Enrico, pour ton aide dans l'apprentissage d'Elpi, tes nombreuses réponses détaillées à mes questions techniques, et ton accueil chaleureux en ~~Italie~~ région niçoise.

Merci à Sandrine et Anders pour votre travail rigoureux dans la relecture de ce document. Merci à tous les membres de mon jury de thèse pour leur présence à ma soutenance et leur intérêt pour mon travail.

Ensuite, je souhaite remercier les autres ex-doctorants, doctorants et futurs doctorants qui ont croisé ma route. Merci à vous tous pour les discussions variées, les pauses café, les soirées au bar.

Merci Martin d'avoir été mon compagnon de route tout au long de cette thèse. Merci pour ton optimisme, ta gentillesse, ton humour et tous les services rendus, dont le fait de m'avoir nourri lorsque j'étais malade et (plus ou moins) coincé dans une résidence étudiante à Édimbourg. Peut-être saurons-nous un jour qui est le vrai thésard d'Assia.

Merci Hamza d'avoir partagé mon bureau pendant des mois et d'avoir fait office de canard en plastique lorsque j'étais bloqué et que j'avais besoin d'une oreille attentive pour comprendre et résoudre mes problèmes.

Merci Xavier, pour ton accueil à mon arrivée, ta grande culture en informatique tant théorique que pratique, tes bons conseils ainsi que ta résistance à toute épreuve à l'angoisse des dates butoirs.

Merci Theo (Winterhalter), Meven et Loïc, pour votre accueil chaleureux, votre patience et votre pédagogie dans le partage de votre connaissance pointue de la théorie des types, ainsi que votre disponibilité pour répondre à mes questions même après votre départ.

Merci Sidney d'avoir été mon compagnon de *trolling*, de coinche et de pinte dans les situations d'urgence.

Merci Pierre, pour ta capacité remarquable à penser constamment en dehors de la boîte, comme on dit outre-Manche.

Merci Nils de m'avoir prouvé que les Allemands n'étaient pas tous méchants. À plus sous le bus.

Merci Théo (Laurent), pour nos discussions fructueuses sur la paramétricité, aussi bien en milieu académique qu'au bar ou dans le train.

Merci Arthur (Correnson) d'avoir été mon binôme de chambre à plusieurs reprises aux JFLA et mon repère rennais au milieu de tant de Parisiens.

Merci à Thomas, Peio, Robin, Simon, Koen, Yee Jian, Yann, Tomas^2, Mara, Arthur, Virgil, Axel, Josselin, Amélie, Davide, Emily et Stéphane.

Je remercie aussi les chercheurs, académiques et industriels, qui m'ont beaucoup appris.

Merci Pierre-Marie, pour ta capacité remarquable (du numéro 3, du medium) à adapter ton vocabulaire à la culture scientifique et technique de ton interlocuteur et ainsi pouvoir expliquer aisément énormément de notions techniques. Merci pour ton humour et ta culture linguistique, politique, cinématographique et musicale, qui ont enchanté mon quotidien au long de ces années passées chez Gallinette. Vive le Luxembourg !

Merci Matthieu (Piquerez), pour ta vision du monde du travail, ton calme et ta façon de travailler sereinement dans l'informatique en continuant d'être un matheux.

Merci Yannick, pour ta grande culture scientifique, ta précision germanique dans le raisonnement et ta manie de demander sans cesse quand aura lieu la fête.

Merci Nicolas, pour tous tes efforts pour faire vivre l'équipe Gallinette, et pour tes explications sur la paramétricité univalente.

Merci Guillaume, pour ta pensée scientifique indépendante et ton habileté à parler au degré 1.5.

Merci du fond du cœur à Alan, sans qui je n'aurais peut-être jamais continué mon chemin dans la recherche.

Merci Florian, de m'avoir apporté des connaissances sur les nombres flottants et de m'avoir raconté autant de faissoleries croustillantes.

Merci David, pour la liberté que j'ai pu avoir pendant ma thèse CIFRE chez Mitsubishi Electric et pour la possibilité de publier nos résultats en *open source*.

Merci à Kazuhiko, Matthieu (Sozeau), Guilhem, Kenji, Marie, Gaëtan, Benoît, François, Éric, Delphine, Frédéric et Thomas.

Merci à Éric (Tanter) et Quentin pour les discussions intéressantes lors de vos visites à Nantes.

Merci à tous les autres chercheurs avec qui j'ai échangé pendant ces trois années.

Merci aux enseignants hors du commun qui ont contribué à la construction de la personne que je suis aujourd'hui : Marie-Jeanne, Marie-Odile, M. André, M. et Mme Hesry et M. Kerambellec.

Merci à mes professeurs de l'INSA, et tout particulièrement Pascal Garcia qui m'a permis de développer ce goût de l'informatique bien faite.

Sur le plan personnel, je souhaite remercier ma famille pour tout l'amour qu'elle m'a apporté. Il s'agit bien du « centre autour duquel tout gravite et tout brille », comme écrivait Victor Hugo.

Merci à mes parents, ma sœur Sarah, mon frère Roman, mes grands-parents, mes oncles et tantes, mes arrière-grands-parents et mes cousins.

Merci à Octave et Julia qui font maintenant partie de la famille.

Merci à ma belle-famille : Jérôme, Nathalie, Alexis et Clarisse.

La rédaction d'une thèse est une rare occasion de pouvoir rédiger un texte qui persistera à travers le temps. Je profite donc de cette occasion pour adresser une pensée chaleureuse à mes potentiels enfants, neveux, nièces, et tous les descendants de ma famille qui pourraient tomber sur ce document un jour.

Je remercie également mes amis pour leurs encouragements et leur présence à mes côtés.

Merci à mes amis de longue date, Martin, Baptiste, Hugo, Timothé, Florent et mon petit hyyu, pour tous ces moments passés ensemble.

Merci à mes collègues de l'INSA de m'avoir si bien accompagné pendant mes études. Merci en particulier à Nominoë, Alexis, François, Gaël, Lucas et Antoine. À nos parties de cidre-pong et aux fous rires qui vont avec. Aux expatriés ou futurs expatriés : que la Bretagne et la France vivent dans vos cœurs, où que vous soyez dans le monde, et à nos retrouvailles.

Merci Thibault, la « pourriture communiste », toujours disponible pour aller boire une bière et refaire le monde.

Mes salutations fraternelles au camarade Franouch.

Merci à Gaël, Antoine, Taha, William, Olivier et Julien, pour nos échanges de conseils au long du chemin du doctorat.

Enfin, un énorme merci à Séverine, celle qui a passé le plus de temps avec moi pendant ces années.

Merci pour ton soutien sans faille.

Merci pour les balades et les vacances qui m'ont permis de changer d'air.

Merci d'avoir ramené Trusty et Umi à la maison, et par la même occasion, énormément d'amour et de bons moments, des sorties variées, de magnifiques photos, et une saine occupation sur notre temps libre.

Merci d'avoir accepté de regarder Star Wars.

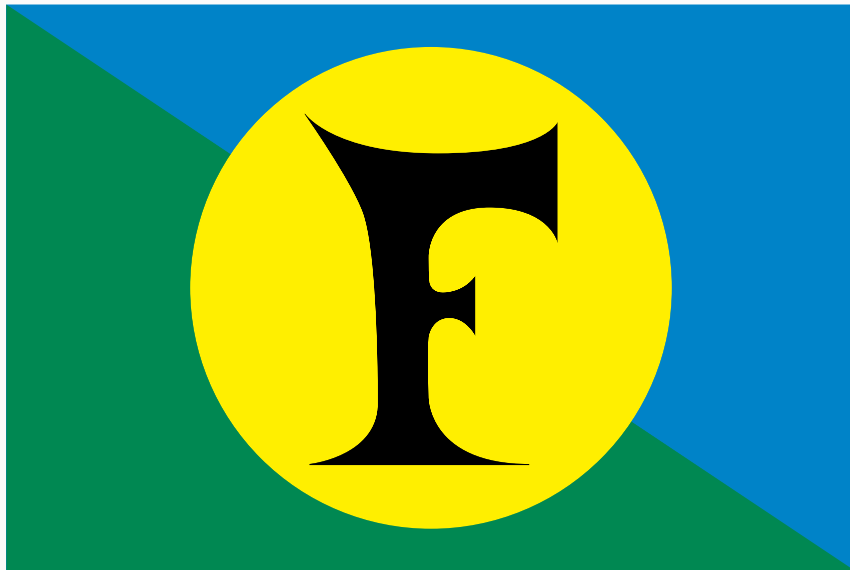
Merci d'avoir accepté que je passe parfois beaucoup de temps avec celle que tu appelles Thérèse et que tu peux à présent voir sous forme écrite et surtout terminée.

Je remercie chaleureusement tous ceux qui vont lire, citer, reprendre et faire vivre cette thèse à l'avenir.

Empire de la Basse Chesnaie

Ministère des Technologies nouvelles

« Ah tiens, d'habitude ça marche. »



Ad Majorem Propriam Gloriam

Contents

Contents

1 Introduction **1**
1.1 A short history of logic . . . 2
1.1.1 Historical foundations . . . 2
1.1.2 Towards mathematics and computer science . . . 3
1.1.3 Mechanisation of logic . . . 3
1.2 Proof assistants and automation . . . 4
1.2.1 A rock-solid reliability . . . 4
1.2.2 Still highly manual proofs . . . 4
1.2.3 Contributions of this thesis . . . 5

THE COQ PROOF ASSISTANT: THEORY AND PRACTICE **6**

2 A short primer to the Coq proof assistant **9**
2.1 Types for proofs and programs . . . 9
2.1.1 The pure λ-calculus . . . 9
2.1.2 Simple types and the Curry-Howard correspondence . . . 10
2.1.3 The Calculus of Constructions . . . 13
2.2 An expressive programming language . . . 14
2.2.1 Universes and polymorphism . . . 14
2.2.2 Inductive types . . . 16

3 Proof assistance **22**
3.1 Inference . . . 22
3.1.1 Unification . . . 22
3.1.2 Inference and *ad hoc* polymorphism . . . 25
3.2 Tactics and automation . . . 26
3.2.1 The proof mode . . . 26
3.2.2 Automated proof tactics . . . 29
3.3 Rewriting and proof transfer . . . 31
3.3.1 Rewriting . . . 31
3.3.2 Extension to equivalence . . . 32

4 Meta-programming in Coq with Coq-Elpi **34**
4.1 A logic meta-programming language for Coq . . . 34
4.1.1 A logic programming legacy . . . 34
4.1.2 Encoding of Coq terms . . . 36
4.2 A toolbox . . . 37
4.2.1 Databases . . . 38
4.2.2 Creation of commands and tactics . . . 39

TRAKT: PROOF TRANSFER BY CANONISATION **41**

5 Goal canonisation: objectives and current situation **43**
5.1 Content of the desired preprocessing algorithm . . . 43
5.1.1 Preprocessing of theories . . . 43
5.1.2 Status of logic . . . 45

5.1.3 Polymorphism and dependent types . . . 46
5.2 The zify family: features and limits . . . 46
5.2.1 Modular preprocessing of arithmetic . . . 47
5.2.2 Preprocessing of logic . . . 48
5.2.3 The mczify extension . . . 49
5.2.4 Limitations of zify . . . 49

6 Theoretical mode of operation **51**
6.1 Gathering user information . . . 51
6.1.1 Type embeddings . . . 51
6.1.2 Logical embeddings . . . 52
6.1.3 Symbol embeddings . . . 53
6.1.4 Conversion keys . . . 53
6.2 Preprocessing algorithm . . . 54
6.2.1 Handling universal quantifiers . . . 54
6.2.2 Handling logical connectives . . . 56
6.2.3 Theory-specific preprocessing . . . 58
6.2.4 The trakt tactic . . . 59

7 Conclusion and perspectives **61**
7.1 Ecosystem of automation tools for Coq . . . 61
7.1.1 The need for preprocessing . . . 61
7.1.2 Modular transformations of the scope tactic . . . 63
7.2 Success of the plugin . . . 64
7.2.1 Examples of goals handled . . . 65
7.2.2 Integration of Trakt with other tools . . . 66
7.3 Paths of improvement . . . 67
7.3.1 Polymorphism and dependent types . . . 67
7.3.2 Architecture of the preprocessing phase . . . 68

Trocq: proof transfer by parametricity **70**

8 Parametricity in dependent type theory **73**
8.1 Motivation and definition . . . 73
8.1.1 Typing and properties of λ-terms . . . 73
8.1.2 Raw parametricity translation . . . 74
8.1.3 Limitations of the raw translation . . . 75
8.2 Univalent parametricity . . . 75
8.2.1 Enrichment of parametricity witnesses . . . 76
8.2.2 Type equivalence and univalence . . . 77
8.2.3 Univalent parametricity translation . . . 79
8.2.4 Omnipresence of the univalence axiom . . . 81

9 Type equivalence in kit **82**
9.1 A new formulation of type equivalence . . . 82
9.1.1 Decomposing equivalence . . . 83
9.1.2 Hierarchical recomposition of parametricity witnesses . . . 86
9.2 Populating the hierarchy of relations . . . 87
9.2.1 Translation of universes . . . 87
9.2.2 Translation of dependent products . . . 88
9.2.3 The case of non-dependent products . . . 89

10 A calculus for proof transfer **90**
10.1 Raw parametricity sequents . . . 90

10.2 Univalent parametricity sequents . . . 92
10.3 Annotated type theory . . . 93
10.4 The Trocq calculus . . . 94
10.5 Constants . . . 96

11 Conclusion and perspectives **98**

Implementation of preprocessing tools with Coq-Elpi **100**

12 Software architecture of a preprocessing plugin **102**
12.1 User knowledge base . . . 103
12.1.1 Use of Coq-Elpi databases . . . 103
12.1.2 Storage of Coq terms . . . 104
12.2 Traversal of the initial goal . . . 104
12.2.1 A translation tactic in Coq-Elpi . . . 104
12.2.2 Trakt: lessons of a first attempt . . . 105

13 Implementation of a parametricity plugin **109**
13.1 Generating and inhabiting the parametricity hierarchy . . . 110
13.1.1 Generation of the hierarchy and plugin set-up . . . 110
13.1.2 Flexibility of parametricity witnesses . . . 112
13.2 Implementation of the parametricity relation . . . 113
13.2.1 From inference rules to a logical program . . . 113
13.2.2 Useful Coq-Elpi features . . . 115
13.3 Parametricity class inference . . . 116
13.3.1 Problem definition . . . 117
13.3.2 Solution chosen in Trocq . . . 119
13.3.3 Implementation . . . 120
13.3.4 Weakening and subtyping . . . 122
13.4 Universe polymorphism . . . 122
13.4.1 Clearing typical ambiguity . . . 123
13.4.2 Algebraic universes and bound universes . . . 124

Conclusion and perspectives **126**

Bibliography

Introduction 1

1.1 A short history of logic 2
1.1.1 Historical foundations 2
1.1.2 Towards mathematics and computer science 3
1.1.3 Mechanisation of logic 3
1.2 Proof assistants and automation 4
1.2.1 A rock-solid reliability 4
1.2.2 Still highly manual proofs . . . 4
1.2.3 Contributions of this thesis . . 5

It is common practice, both in academia and industry, to use various methods of *verification* of results, with the aim of increasing society's confidence in these results. Indeed, if the proof of a mathematical theorem is understood and accepted by its author's community, it can then be taken for granted and reused in subsequent research work. In an applied environment, engineers can also use theoretical results in the design of concrete industrial products. The various *tests* carried out on these products before they leave the factories are another way to ensure the confidence of future customers and users. More generally, the goal of verification is to erase the naturally imperfect aspect of human work, and to enable each generation to tackle increasingly complex problems, thanks to the work of the previous generation, which it trusts.

> *Errare humanum est.*[1]
>
> Roman maxim

1: "To err is human."

Unfortunately, despite all the verification procedures in effect, on many occasions errors have crept into works, whether they be mathematical proofs or computer programs. On June 4th, 1996, for instance, the Ariane 5 rocket ended its maiden flight[2] after about thirty seconds with an explosion caused by a computer bug called *integer overflow*,[3] causing a loss of hundreds of millions of euros. More recently, in 2018 and 2019,[4] two Boeing 737-MAX planes crashed within minutes of take-off because software designed to prevent stalls failed and overrode manual control. This time, the death toll was extremely high, amounting to several hundred people.

2: Flight 501.

3: On a machine, numbers are represented by a binary value of a certain fixed size so that they can be stored in memory. In this way, they cannot exceed a certain maximum value. An *integer overflow* occurs when the result of an arithmetic operation exceeds this value. The binary value retained for the number then becomes very small, which is often a source of errors.

4: Flights Lion Air 610 and Ethiopian Airlines 302.

> Software testing shows the presence, not the absence of bugs.
>
> Edgser W. Dijkstra (Dutch computer scientist)

The obvious conclusion is that testing, *a fortiori* carried out by humans, is no longer enough, and that the time has come for *formal* verification, *i. e.*, the use of tools to *certify* the absence of errors in a mathematical proof or in a computer program. This is the promise of *formal methods*, a field of research that provides a theoretical basis for carrying out proofs on computers. In this vein, the end of the 20th century saw the emergence of a family of software called *Automated Theorem Provers* (ATPs). These are digital implementations of proof search algorithms in a given logical theory, allowing a human to enter a statement to be proved and let the computer determine whether it is true or false.

> *Quis custodiet ipsos custodes?*[5]
>
> Juvenal (Roman poet)

5: "Who watches the watchers?"

However, there is still a concern that these provers are not themselves infallible, since they are derived from code written by humans, and may therefore contain errors. *Interactive Theorem Provers* (ITPs), the flagship of formal methods, are a response to this problem. This family of software is designed around a *logical core*, a small amount of code that directly implements the rules of a logical theory and is trusted by the users. Various tools are made available to the user to carry out proofs, each proof being eventually verified by the kernel, guaranteeing unfailing confidence at all times in all developments carried out using such

software. Conceptually, it is an optimal trade-off between human and machine. Indeed, since a computer is not really capable of producing *original* reasoning,[6] this task is left to humans. Yet, the machine excels in the mechanical application of the logical rules of the kernel to check that a proof is correct, whereas a human could make mistakes.

6: Despite the recent impressive results in this field of research, machine learning essentially consists of compiling and harnessing a huge amount of information in the best possible way. This information may be far more massive than the knowledge of a single human being, but it is not a question of giving the machine the human traits of originality, creativity, *etc.*

The price to pay for this additional safety is the interactive nature of these proof assistants. Indeed, in order to represent highly abstract computer programs and mathematical theories within a proof assistant and guarantee their versatility, the underlying logical theory is often much more complex than in an automated theorem prover. Furthermore, in order to have a readable and trustworthy kernel, proof assistants do not benefit from the aggressive heuristics and optimisations present in the code of ATPs. As a result, the user often has to spell out uninteresting details in the proofs, and any automation becomes an arduous task.

This thesis is part of a research effort towards proof automation, to facilitate the work of users of proof assistants, the ultimate goal being to spread the use of these tools in place of software testing, wherever this is possible and relevant. It therefore lies on the borderline between computer science and mathematics, and oscillates between theoretical contributions and implementation work, because it is important to provide users with tangible tools as quickly as possible. To place this work in a broad context, in this chapter we briefly retrace the history of logic (§ 1.1) before presenting proof assistants along with their current level of automation, as well as the major contributions (§ 1.2).

1.1 A short history of logic

Logic is the study of the formal rules used to determine whether a line of reasoning is valid. This discipline was founded in Antiquity. More recently, it got closer to mathematics then theoretical computer science, by coming in the shape of formal logic systems. Finally, with the advent of computers, logic became mechanised. In this section, we give some details of these different stages.

1.1.1 Historical foundations

In the West, the founding work in the field of logic dates back to ancient Greece. Indeed, this discipline, then called *λογική*, held a central place in public life there, and many concepts still used today in this field come from thinkers of that time, notably Aristotle and Euclid.

In his *Organon*, Aristotle defines the structure of logical reasoning. In this work, he differentiates between the notions of *being* and *predicate*,[7] *cause* and *consequence*, or *affirmation* and *negation*. He also introduces a logical construction called *syllogism*, linking two premises and a conclusion by deduction.[8]

7: The *beings* are the entities and the *predicates* represent what can be said about them.

8: The best-known syllogism is probably the following:

- All men are mortal;
- Socrates is a man;
- Therefore, Socrates is mortal.

For his part, Euclid introduces definitions associated with demonstration, such as *postulates* or *axioms*, unproved hypotheses that are taken as the basis of a logical system, or *propositions* which are statements that can be proved. To these were added numerous theorems, particularly in geometry and number theory, to form the work of the *Elements*, which went on to become a veritable academic reference. In general, until the end of the Middle Ages, logic was taught with these founding works.

1.1.2 Towards mathematics and computer science

After centuries of unprecedented scientific advances throughout the world, the 19th century was marked by a quest to formalise logic. Indeed, the aim was to build a common language for mathematics. In this context, *formal* languages[9] were gradually developed. For example, FREGE introduced in the *Ideography* [1] the concept of *quantification*[10] and the *predicate calculus*,[11] which are still used today. PEANO proposed an axiomatisation of arithmetic based on natural numbers [2], which gave birth to the *mathematical induction* reasoning taught in mathematics nowadays. Finally, CANTOR created *set theory* [3], which made it possible to describe all the mathematical objects of his time within a common framework. Logic, although traditionally a discipline close to philosophy, became a branch of mathematics.

9: Formal languages are defined by a syntax, *i. e.*, a finite set of symbols that can be used to create formulas according to precise and explicit rules. These symbols can then be given a logical interpretation and rules for reasoning can be defined, which are also explicit, so as to leave no ambiguity when manipulating the language in this context.

[1]: FREGE (1882), "Begriffsschrift : Eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens"

10: He introduces two *quantifiers*:

- the universal quantifier — "for all x";
- the existential quantifier — "there exists x".

11: Also known as *first-order logic*.

[2]: PEANO (1889), "Arithmetices principia : Nova methodo exposita"

[3]: CANTOR (1883), "Grundlagen einer allgemeinen Mannigfaltigkeitslehre. Ein mathematisch-philosophischer Versuch in der Lehre des Unendlichen"

During the 20th century, the theory of formal languages brought programming languages, used to implement *algorithms*.[12] The rules associated with these languages are then *computation* rules enabling programs to be actually executed to obtain a result. However, several scientists identified links giving these languages a logical interpretation: this parallel is known as the CURRY-HOWARD correspondence. This discovery marked a convergence between logic and the emerging theoretical computer science.

12: An algorithm is a sequence of operations to be performed in order to solve a particular problem.

1.1.3 Mechanisation of logic

The development of computers allowed putting into practice the various formal systems studied previously. This step heralded a new era for logic and mathematics, in which humans and machines would work together. The PROLOG [4] system was born in the early 1970s. In this programming language, the developer defines a base of facts and rules for statements to be valid, and the user asks questions to the system. PROLOG has been used extensively for language processing, but the ability to reason by induction over numbers and trees also makes it suitable for processing logical formulas. In this way, the user's conjectures can be expressed as queries, the validity of which the system can check by executing them.

[4]: COLMERAUER *et al.* (1973), "Un système de communication homme-machine en français"

Implementations of logical systems also include the family of automated theorem provers, whose most widespread representative is the SAT[13] solver. It receives as input a formula in *propositional logic*[14] and determines if there exists a *valuation*[15] that makes it true. SAT solvers are very popular because of the wide variety of problems they can represent: planning, electronic circuit design, cryptanalysis, *etc.* By adding symbols and specific behaviour for one or more theories, such as arithmetic or bit vectors, we get an SMT[16] solver. This extension widens the scope of SAT solvers to program verification problems.

13: Boolean satisfiability.

14: Formed from variables and negation, conjunction or disjunction connectives.

15: Assigning a truth value (true or false) to each variable in the formula.

16: Satisfiability Modulo Theory.

Tools dedicated to certification of algorithms and mathematical theorem proving were subsequently developed, such as *model checking* tools like TLA+ [5]. Such software allow users to represent the objects they wish to reason about and to describe the statements they wish to prove, all in an expressive formal language closer to natural language. For example, these tools can be used to prove the correctness of concurrent and distributed algorithms, or to prove that an automaton can never be in an invalid state (for example, a lift at a standstill between two floors). Certain tools such as WHY3 [6] or LIQUID HASKELL [7] allow programs to be written in a language from the ML family and the various statements to be proved directly in the same file. Within this framework, part of the proof burden is delegated to SMT solvers, offering a certain level of automation, in such a way

[5]: LAMPORT (1994), "The Temporal Logic of Actions"

[6]: FILLIÂTRE *et al.* (2013), "Why3 — Where Programs Meet Provers"

[7]: VAZOU (2016), *Liquid Haskell: Haskell as a Theorem Prover*

that for certain use cases, this family of tools is currently considered to be a good compromise.

1.2 Proof assistants and automation

Despite the level of automation provided by an SMT solver, these tools are often used as black boxes, often unable to explain their reasoning steps. The confidence that can be placed in these tools therefore remains limited, especially as errors are sometimes discovered in their highly optimised code yet difficult to update in a correct way. In the most critical applications, the extra level of reliability required is provided by *proof assistants*, software with a more radical approach but more manual for the user. In this section, we present these different aspects of proof assistants before listing the major contributions of this thesis.

1.2.1 A rock-solid reliability

Proof assistants, or interactive theorem provers, are software designed to perform proofs based on collaboration between humans and machines. This kind of software is built around a *logical kernel*, a small amount of code representing the rules of the logical system implemented by the proof assistant. This kernel is the only trusted code base for the user, as it verifies all the proofs performed in the proof assistant. Each proof validated by the kernel can then be used in other proofs, making it possible to progressively build entire libraries of proofs in a given domain.

Proof assistants can be based on a wide range of logical systems. For example, the Isabelle/HOL [8] proof assistant is based on higher-order logic. In this thesis, we are interested in a family of proof assistants based on type theory, an expressive language, result of a thorough exploitation of the Curry-Howard correspondence, used to programme, express the statements to be proved, and carry out the proofs, all in one. Examples of software in this family are Agda [9], Lean [10] and Coq [11], the proof assistant on which this work is based. In this framework, proofs are represented by proof terms, and the kernel actually checks them by calling the typechecker[17] of the underlying language.

[8]: Nipkow *et al.* (2002), *Isabelle/HOL: a proof assistant for higher-order logic*

[9]: Norell (2008), “Dependently Typed Programming in Agda”

[10]: De Moura *et al.* (2021), “The Lean 4 Theorem Prover and Programming Language”

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

17: On a computer, all values are represented in binary. However, it would be impractical to write complex programmes that only manipulate numbers. In order to raise the level of abstraction, many programming languages encode various data structures in binary while offering the developer a way of manipulating them in a different way than numbers, by giving them a *type*. One can then manipulate strings of characters, lists, *etc.* Typed languages are then equipped with a typechecker to validate that the operations associated with one type are not used on a value of another type, which would cause the programme to lose its meaning.

1.2.2 Still highly manual proofs

The downside of this level of confidence is that proofs once again become manual, as the proof assistant requires all the steps in a proof to be made explicit by a proof term. However, although the language in the Coq proof assistant is very expressive, it is still very different from the natural language used by humans, and it is difficult for a user to enter proof terms into the software by hand. Moreover, some proof steps need to be made explicit without being interesting for the user. For example, explaining to the proof assistant mathematical manipulations that are trivial on paper, such as commutativity of addition,[18] takes a substantial amount of time and slows down the user’s proof work.

18: $a + b = b + a$.

In order to solve these problems, the piece of software provides a toolbox bringing its formal language closer to the natural language used in paper proofs. In particular, these tools include inference functions, whereby the user can supply

incomplete terms and let the machine determine the missing pieces. For example, the user does not need to make explicit the types of all the values, and the proof assistant can define notations so that the terms entered by the user are similar to the notations they would use in a paper proof. In addition, proof automation tools are made available to the user to prove in a few commands a specific class of statements corresponding to the stages of the proof on which an expert mathematician does not wish to spend most of his time. These tools are what make such a piece software a real proof *assistant*.

1.2.3 Contributions of this thesis

Proof mechanisation is only possible if automation solutions are made available. One possible approach is to link proof assistants to automated theorem provers, widely used in formal methods, in order to facilitate the proof of statements within their reach. However, this requires formulas used on both sides to correspond to each other, aligning logic, data types, operations, *etc.* More generally, the problem of *proof transfer*, on the scale of a single logical formalism, is the problem of expressing the same mathematical concept in several different ways, without any impact on the proofs, *i. e.*, when a proof has been carried out using a representation of a mathematical concept within the proof assistant, we do not want to have to redo this proof manually using another representation of the same concept. With the aim of solving an instance of the proof transfer problem in the Coq proof assistant, we propose Trakt [12], a preprocessing tool for Coq statements, that makes the statements of a given theory converge to a canonical form in the ideal format expected by a proof automation tool for this theory.

[12]: Blot *et al.* (2023), "Compositional preprocessing for automated reasoning in dependent type theory"

Other properties of the formal languages used in proof assistants can be exploited for proof transfer, such as *parametricity* [13]. It is an interpretation of types as relations, enabling to build tools that link a Coq statement with an associated statement that is the target of the proof transfer. In rich versions of parametricity, such as *univalent parametricity* [14], one can extract a proof term from the witness[19] of the relation between both statements, and exploit it to concretely carry out the proof transfer. However, univalent parametricity introduces axioms into Coq in order to work correctly, including cases in which a manual processing using no axioms would be possible. We then present Trocq [15], a second proof transfer plugin for Coq, more general than Trakt, aiming to match the power of univalent parametricity in as many cases as possible, while analysing the statement more finely and using axioms in a smaller number of cases.

[13]: Reynolds (1983), "Types, Abstraction and Parametric Polymorphism"

[14]: Tabareau *et al.* (2021), "The marriage of univalence and parametricity"

19: The *witness* of a relation R between two values a and b is a proof that these two values are linked in the relation, *i. e.*, a proof of $R\,a\,b$.

[15]: Cohen *et al.* (2024), "Trocq: Proof Transfer for Free, With or Without Univalence"

These two tools take the form of plugins for the Coq proof assistant. As such, they rely on particular meta-programming techniques and their implementation raises specific questions independent of their theoretical design. The chosen meta-language for these implementations is Coq-Elpi [16], a logic programming language that offers a high level of abstraction in the manipulation of Coq terms.

[16]: Tassi (2018), "Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect)"

The remainder of this thesis is organised in four parts: a technical introduction defining the various concepts handled in the subsequent parts (§ I), a presentation of both prototypes of preprocessing tools developed, Trakt (§ II) then Trocq (§ III), as well as a section dedicated to implementation issues (§ IV).

The Coq proof assistant:
theory and practice

Introduction

Formal methods are a major tool to increase the level of confidence that one can put in computer programs. By introducing rigorous reasoning, they provide more guarantees than more traditional software quality assurance methods such as software testing, and are very popular in the development of critical software. Of the many formal tools available, those with the most radical approach are proof assistants, because in such software, the proofs themselves are guaranteed to be formally correct. At the price of trusting a logical kernel made up of a few hundred lines of code implementing the logical system at the heart of the proof assistant, such software can then be used to check the compliance of programs with a specification or the validity of a mathematical proof.

Within a proof assistant, a programming language is used to describe the mathematical objects on which we wish to reason[20] and a proof language is used to prove, step by step, statements expressed using the various data structures defined beforehand. This thesis focuses on the Coq [11] proof assistant, developed in France in the 1980s. In this piece of software, the expressiveness of the programming language used is such that it is also possible to carry out proofs in this same language, including verification of proofs in the typechecker. This expressiveness is a real advantage for the user, first because they only have a single language to trust, but also because it is possible to represent very abstract mathematical objects in Coq and to reason about the associated theories, which is not the case of all formal tools. However, this expressiveness makes proofs very difficult to carry out, as they are represented with proof terms in this very advanced language, whose very high level of abstraction in programming tasks becomes a very low level of abstraction in the elaboration of proofs.

20: For example, a computer program is represented by an abstract syntax tree obtained from its source code.

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

Proof automation in Coq is therefore the motivation behind numerous research projects in computer science. Their main aim is to build a layer of abstraction above the programming language of Coq, so that the user does not have to write proofs manually in this language. The proof assistant then has a tactic language allowing the proof to make progress step by step, by executing actions that are intuitive for the user and authorised by Coq following verification of a fragment of the final proof term, provided by the tactic. These tactics stepping from one proof state to another are implemented at the meta level, using one of the meta-languages available in Coq.

Some tactics implement a decision procedure, *i. e.*, an algorithm able to decide whether a statement contained in a theory is true or false, by automatically constructing the associated proof term. For example, the `lia` [17] tactic can automatically prove a true statement that belongs to the theory of Presburger arithmetic. Other tactics reformulate the statement to prove, in order to make it simpler to prove manually or more suitable for other automation tactics. Among the latter are proof transfer tools, that allow proofs or statements expressed using one encoding of a mathematical object to be reformulated using another encoding of the same object, that the user deems more suitable for the proof they are currently performing. Proof transfer is another kind of abstraction in proofs, allowing to erase the differences inherent in the fact of encoding the same mathematical object in several different ways in a proof assistant.

[17]: Besson (2006), "Fast Reflexive Arithmetic Tactics the Linear Case and Beyond"

The main contributions of this thesis are the design of two preprocessing tools for Coq statements, each responding from a specific angle to the problem of proof

transfer. This part provides the technical definitions needed to follow the detailed presentation of these tools. We present the Coq proof assistant, the real proof assistance features it implements, as well as the Coq meta-programming plugin used throughout this thesis, Coq-Elpi [16].

[16]: Tassi (2018), “Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect)”

A short primer to the Coq proof assistant 2

In this chapter, we briefly present the language of Coq in order to introduce the concepts we refer to in the rest of this manuscript. We first present in a gradual way the core of the type theory implemented by the kernel and the associated logical theory (§ 2.1), then we present the features Coq adds on top of this theoretical basis (§ 2.2).

2.1 Types for proofs and programs 9
2.1.1 The pure λ-calculus 9
2.1.2 Simple types and the Curry-Howard correspondence . . . 10
2.1.3 The Calculus of Constructions 13
2.2 An expressive programming language 14
2.2.1 Universes and polymorphism . 14
2.2.2 Inductive types 16

2.1 Types for proofs and programs

The proof assistant Coq is built around a kernel whose essential component is the typechecker of a dependently-typed λ-calculus. This piece of software allows the user to declare custom types called inductive types, with which they can represent the mathematical objects they wish to reason about. The language of Coq can also be interpreted as a logical system, making it a proof language. This section studies these different aspects.

2.1.1 The pure λ-calculus

Introduced in the 1930s by Church [18], λ-*calculus* is a minimalist programming language whose syntax defines only three classes of terms: variable, abstraction, application.

[18]: Church (1933), "A set of postulates for the foundation of logic"

$$t, u \quad ::= \quad x \mid \lambda x.\, t \mid t\, u$$

The base object of such a language is the function, represented by the case of the abstraction $\lambda x.\, t$ where x is a *bound* variable[1] and t is the body of the function. What makes this language a calculus is the β-reduction rule, that allows computing the application of a function to an argument[2] by substituting the argument for the function's bound variable:

1: A *bound* variable, as opposed to *free* variables, only has meaning inside a function. It is used to represent the future argument that will potentially be given to this function when it is applied, by binding this argument to a name.
2: Such a term is called a β-*redex*.

$$(\lambda x.\, t)\, u \quad \rightsquigarrow \quad t[x := u]$$

Substitution is defined in such a way as to avoid capture — the fact that a variable that is free before substitution becomes bound after substitution — as this would give a term with a different meaning from the expected term. Two λ-terms that differ only in the name of their bound variables have the same behaviour with respect to β-reduction. They are then said to be α-equivalent. We can define an equivalence between terms called *conversion*[3] by the transitive symmetrical reflexive closure of β-reduction and α-equivalence.

3: We denote $t \equiv u$ for "t is convertible to u".

Many programming concepts can be encoded using λ-terms: integers, booleans, pairs, lists, trees, *etc.* It is even possible to encode recursion using fixed-point combinators, like the Y combinator:

$$Y \quad := \quad \lambda f.\, (\lambda x.\, f\, (x\, x))\, (\lambda x.\, f\, (x\, x))$$

Indeed, $Y\ f$ reduces to $f\ (Y\ f)$, then $f\ (f\ (Y\ f))$, and so on indefinitely. It is in fact possible to encode any Turing-computable program in a λ-term.[4]

4: Turing-computability of a program corresponds to the possibility to encode it on a Turing machine. Turing himself proved that λ-calculus is equivalent to his machine [19], *i. e.*, that it is Turing-complete.

Reducing a term is done by applying the β-reduction rule as much as possible in this term. The path taken while reducing the term is thus not necessarily unique. If a reduction path arrives to a term containing no more redex, *i. e.*, it cannot be β-reduced further, this final term is said to be a *β-normal form*. The property called *normalisation* is the existence of a normal form for any term in the calculus.[5] As the Y combinator shows, λ-calculus does not respect this property because it allows encoding *general recursion*, *i. e.*, arbitrary loops.

5: If all the reduction paths lead to a normal form, the calculus respects the property of *strong normalisation*.

2.1.2 Simple types and the Curry-Howard correspondence

In order to obtain several interesting properties, including normalisation, we can restrict the set of terms that can be defined in λ-calculus. We then introduce *types* [20], *i. e.*, annotations on terms, with *typing rules* enabling us to infer or verify these types, and by extension to describe the terms we wish to authorise in the calculation, an ill-typed term being rejected.

[20]: Church (1940), "A formulation of the simple theory of types"

Definition of simply-typed λ-calculus The function being the base object of λ-calculus, a type is defined as either a variable α belonging to a finite set of base types, or a functional type from a domain to a codomain using an arrow:

$$\tau \quad ::= \quad \alpha \mid \tau \to \tau$$

Then, we annotate functions by adding a type to their bound variable:

$$t, u \quad ::= \quad x \mid \lambda x : \tau . t \mid t\, u$$

A term can only be well typed in a typing context, *i. e.*, a list of associations between variables and types:

$$\Gamma \quad ::= \quad \langle\rangle \mid \Gamma, x : \tau$$

A typing judgment $\Gamma \vdash t : \tau$ asserts that term t has type τ in context Γ. In order to obtain a typing judgment on a term t, one composes typing rules together by induction on the syntax of t, building a *typing derivation* whose conclusion is the typing judgment on term t. The typing rules for this calculus called *simply-typed* λ-calculus and denoted $\lambda_\to$ are then the following:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (\textsc{Var}_\to) \qquad \frac{\Gamma, x : \tau \vdash t : \tau'}{\Gamma \vdash \lambda x : \tau . t : \tau \to \tau'}\ (\textsc{Lam}_\to)$$

$$\frac{\Gamma \vdash t : \tau \to \tau' \qquad \Gamma \vdash u : \tau}{\Gamma \vdash t\, u : \tau'}\ (\textsc{App}_\to)$$

Figure 2.1: Typing rules for $\lambda_\to$

The $\lambda_\to$ calculus can be used to write sensible programs whose results can be obtained by computing their normal form. For example, one can add to the calculus a base type $\mathbb{N}$ to represent natural numbers, as well as two constants $0 : \mathbb{N}$ and $S : \mathbb{N} \to \mathbb{N}$ to represent zero and the successor, *i. e.*, the base components of Peano arithmetic. By using these values, one can then build programs manipulating natural numbers.

The Curry-Howard correspondence By associating the arrow $\rightarrow$ of the functional type with logical implication, simply-typed λ-calculus actually corresponds to the logical system of natural deduction. This crucial link between logic and programming is called the Curry-Howard correspondence. Indeed, by considering base types as propositional variables, simple types can be seen as statements and typing rules as rules of the associated logical system. The typing context corresponds to a set of hypotheses, with free variables and constants being axioms — properties considered as proved before any demonstration begins. The $\text{Var}_{\rightarrow}$ rule describes the case of hypotheses: if a property is in the context, then it is provable. The $\text{Lam}_{\rightarrow}$ rule makes it possible to introduce an implication by noting that if τ' is provable from a proof of τ in a context, then in this context τ implies τ'. Furthermore, the $\text{App}_{\rightarrow}$ rule is the famous logical rule of *modus ponens*: if proposition τ is provable and τ implies τ', then τ' is also provable. If we augment $\lambda_{\rightarrow}$ with additional constructions such as the product[6] or the sum,[7] we can extend the correspondence to propositional calculus. Indeed, a type $\tau_1 \times \tau_2$ corresponds to a conjunction of two propositions, and a type $\tau_1 + \tau_2$ corresponds to a disjunction. Finally, false — also denoted as $\bot$ — is associated to what cannot be proved, *i. e.*, types without any inhabitants.

6: The product type allows us to represent pairs of values. A value of type $A \times B$ is the combination of a value of type A with a value of type B. This is the basic building block for representing the concept of tuple present in many programming languages in a theoretical way. Its constructor is the following:

$$(\cdot, \cdot) : A \rightarrow B \rightarrow A \times B$$

7: The sum $A + B$ is a construct that contains a value of one type from two possibilities A and B. In languages with algebraic types, it is used to perform case analyses, for example between a valid result and an error — `result` in OCaml, `Either` in Haskell, *etc.* Traditionally, a sum is constructed by one of the following constants:

$$\text{inl} : A \rightarrow A + B$$
$$\text{inr} : B \rightarrow A + B$$

Moreover, this correspondence between statements and types is also a correspondence between proofs and programs. In a function $\lambda x : \tau . t$ of type $\tau \rightarrow \tau'$ that constructs an inhabitant of τ' from an inhabitant of τ, the terms x and t are therefore proof terms. A proved statement corresponds to the existence of a term that inhabits the type of this statement. The constants added to the global context are then axioms. In fact, a constant $k : \tau$ is a variable k that we assume inhabits the type τ. The statement corresponding to τ is therefore proved, but by a witness that has no computational content. The analogy naturally extends to other constructions. Thus, a term that inhabits a product type $\tau_1 \times \tau_2$ is a pair of proofs (t_1, t_2) where t_1 is a proof of τ_1 and t_2 is a proof of τ_2, and a term that inhabits a sum type $\tau_1 + \tau_2$ is either $\text{inl}\, t_1$ where t_1 is a proof of τ_1, or $\text{inr}\, t_2$ where t_2 is a proof of τ_2.

The Barendregt cube Simply-typed λ-calculus is the starting point of a handful of generalisations. Indeed, it benefits from a number of interesting properties, including preservation of typing by reduction,[8] uniqueness of typing, and *logical consistency*.[9] In particular, its type system rejects any terms whose reduction does not terminate, such as fixed-point combinators. For instance, the definition of the Y combinator applies a subterm x to itself, which is impossible in $\lambda_{\rightarrow}$, as x cannot have both a functional type $\tau \rightarrow \tau'$ and the type of its domain τ. However, it lacks the expressiveness of pure λ-calculus. For example, in pure λ-calculus, the term $\lambda x. x$ represents the identity function, and can be applied to any term. In simply-typed λ-calculus, every function is defined over a single type. If we want to apply it to terms of different types, we need to build a distinct instance of identity for each type used. Various extensions of $\lambda_{\rightarrow}$ were designed in the second half of the 20th century, with the goal of obtaining a more expressive programming language or logical system, both points of view being available thanks to the Curry-Howard isomorphism.

8: If a term $t : \tau$ reduces to a term t', then $t' : \tau$ — also called *subject reduction*.

9: It is impossible to prove $\bot$ in the logical system from an empty context.

In 1991, Barendregt [21] proposed to represent these extensions as the edges of a cube[10] taking $\lambda_{\rightarrow}$ as its origin and introducing a different form of abstraction on each axis. These forms of abstraction can be described using the concept of *sort*, *i. e.*, the type of a type. We introduce the $\star$ sort — called *type* — and declare that all simple types have $\star$ as their sort. For example, $\mathbb{N} : \star$ and $\mathbb{N} \rightarrow \mathbb{N} : \star$. This gives us a way to characterise more abstract constructions. For example, a term

[21]: Barendregt (1991), "Introduction to generalized type systems"

10: Thus called λ-cube or Barendregt cube.

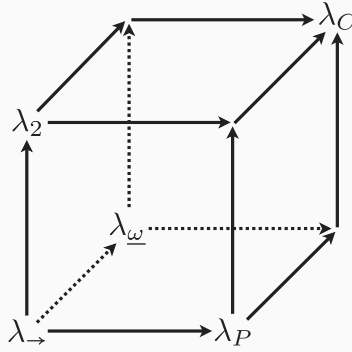


Figure 2.2: The Barendregt cube.

constructing a type from another type would have sort $\star \rightarrow \star$. We introduce the $\square$ sort — called *kind* — as the sort of all sorts. For example, $\star : \square, \star \rightarrow \star : \square$ and $\mathbb{N} \rightarrow \star : \square$. We then identify the different forms of abstraction of a λ-calculus by allowing one or more of the following classes of functional types in the calculus, defined by a pair of symbols chosen from $\star$ and $\square$, representing the sort of the domain and codomain of the authorised functional type:

- Class $(\star, \star)$ describes terms that depend on terms. For example, this is the case for the addition of the natural numbers $+_{\mathbb{N}} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$. This class of terms is definable in $\lambda_{\rightarrow}$ and all its extensions.
- Class $(\square, \square)$, authorised in calculus $\lambda\underline{\omega}$ and its extensions, describes the types that depend on types. These are *type constructors*. For example, we can imagine a term list used to describe the type of a list when it is provided with the type of the elements in the list. Thus, $\mathrm{list}\ \mathbb{N}$ is the type of lists of natural numbers.
- Class $(\square, \star)$, authorised in calculus λ_2 and its extensions, describes terms that depend on types. These are *constructors* of terms belonging to polymorphic types. For example, we can imagine a constant cons that adds an element to a list. This constant depends on the type of the elements in the list.
- Class $(\star, \square)$, authorised in calculus λ_P and its extensions, describes types that depend on terms. This abstraction is that of *dependent types*, the least common in functional programming languages. It can be used, for example, to construct a fixed-length integer array type using a constant narray that depends on an integer n describing the size of the arrays that will have type $\mathrm{narray}\ n$.

All these abstractions can then be represented on three axes using a cube illustrated on Figure 2.2.[11] By generalising this cube to calculi with more than two sorts, we obtain the family of *Pure Type Systems* [22, 23].

11: We represent on the cube only the languages cited here.

[22]: Berardi (1988), "Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube"

[23]: Terlouw (1989), "Een nadere bewijstheoretische analyse van GSTT's"

Each form of abstraction adds expressiveness to the language but potentially weakens its theoretical guarantees. Functional languages that have been concretely implemented and are used at an industrial level, such as Haskell, can hardly be placed on the cube, as their type system includes numerous pragmatic features that either do not necessarily correspond to an end of the cube — Generalised Algebraic Data Types (GADT), Hindley-Milner-style polymorphism, *etc.* — or stray away from a logical interpretation — non-termination. However, their theoretical basis is a restriction of λ_2, preserving decidability of typing and complete inference for practical purposes. As for the Coq proof assistant, it contains an implementation of the language corresponding to the point of the cube lo-

cated furthest from the origin, λ_C or the Calculus of Constructions (CoC)[12] [24]. It therefore has all the abstractions described previously, while retaining a consistent logical interpretation, that of the intuitionistic predicate calculus.

12: Abbreviation partly responsible for the name of this proof assistant.

[24]: Coquand *et al.* (1986), "The calculus of constructions"

2.1.3 The Calculus of Constructions

The syntax of the Calculus of Constructions adds two constructions to that of simply-typed λ-calculus, the dependent product or Π-type and the universe:

$$A, B, t, u \quad ::= \quad x \mid \lambda x : A.\, t \mid t\, u \mid \Pi x : A.\, B \mid \square$$

Dependent product The dependent product is used to describe dependent functional types. Indeed, in the $A \to B$ arrow type of simply-typed λ-calculus, the codomain B does not itself depend on domain A, these terms being both defined outside this functional type. In a dependent λ-calculus, we use a dependent product $\Pi x : A.\, B$ which is a binder in the same way as an abstraction. Thus, just as the body t of a function $\lambda x.\, t$ is defined as a function of the argument x that will be supplied to it when it is applied, the codomain B of a dependent product can depend on the bound variable x over which it quantifies.[13] Indeed, the logical interpretation of the dependent product is universal quantification. For example, if we denote as $\mathrm{narray}\ n$ an array of natural numbers of size n, the term constructing an array of size n by repeating a given value has type:

13: When B does not depend on x, naming the variable is unnecessary, and we can use notation $A \to B$ as syntactic sugar for:

$$\Pi_ : A.\, B$$

$$\mathrm{nreplicate} \quad : \quad \Pi n : \mathbb{N}.\, \mathbb{N} \to \mathrm{narray}\ n$$

The $\mathrm{nreplicate}$ term therefore has type $\mathbb{N} \to \mathrm{narray}\ n$ *for all* integer n supplied as the first parameter. The term $\mathrm{nreplicate}\ 4\ 0$ represents an array of type $\mathrm{narray}\ 4$ filled with zeros, $\langle 0, 0, 0, 0 \rangle$. This quantifier takes on its full logical meaning when used in the type of properties to prove:

$$\mathrm{natpos} \quad : \quad \Pi n : \mathbb{N}.\, n \geq 0$$

Universes and universe hierarchy Note that types are no longer a syntactic category in their own right, but terms like any others, and as such can be manipulated as first-class values in the language. This is a feature of languages with dependent types. Because they are terms like any others, it must be possible to write types to the left of a typing judgment. This is the reason for the existence of the universe $\square$, which is a sort, *i. e.*, the type of types. In this way, we can generalise the array type narray and the associated function $\mathrm{nreplicate}$ to polymorphic arrays:

$$\begin{aligned} \mathrm{array} \quad &: \quad \square \to \mathbb{N} \to \square \\ \mathrm{replicate} \quad &: \quad \Pi A : \square.\, \Pi n : \mathbb{N}.\, A \to \mathrm{array}\ A\ n \end{aligned}$$

As the types of a dependent λ-calculus are also terms, the $\square$ sort can itself be on the left of a typing judgment. However, allowing $\square : \square$ breaks the logical consistency of the theory: this is the Girard paradox [25], the type-theoretic equivalent of the Russell paradox[14] of set theory, according to which there can be no set containing all sets. One solution is to use an augmented version of the Calculus of Constructions called CC_ω, in which every universe $\square_i$ is annotated with a natural number i representing its *level*.[15] We can thus safely postulate that each

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

14: Russell initially explained the paradox in a letter sent to Frege in 1902. Later, Frege published the contents of the letter [26].

15: In the rest of this thesis, for the sake of readability, when a universe level is not important in the discourse, we shall leave it implicit and write $\square$.

universe is included in the next one, hence the name of *universe hierarchy*:

$$\frac{}{\Gamma \vdash \Box_i : \Box_{i+1}}$$

Typing rules The typing rules for CC_ω are available in Figure 2.3. The Lam rule differs from its simply-typed version in that the type of t can now depend on variable x, so the abstraction is typed by a dependent product. For the same reason, the App rule now performs a substitution in the type of an application, in order to instantiate the bound variable x in the codomain B.

$$\frac{}{\Gamma \vdash \Box_i : \Box_{i+1}} \text{(Sort)} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{(Var)}$$

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

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

$$\frac{\Gamma \vdash A : \Box_i \qquad \Gamma, x : A \vdash B : \Box_j}{\Gamma \vdash \Pi x : A.\, B : \Box_{\max(i,j)}} \text{(Pi)} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash t : B} \text{(Conv)}$$

Figure 2.3: Typing rules for CC_ω

2.2 An expressive programming language

Several proof assistants, such as Coq or Lean, are based on close variants of the Calculus of Constructions, but make different choices regarding the details of their underlying formalism. This section details the major choices implemented in Coq concerning universes and inductive types.

2.2.1 Universes and polymorphism

Although initially introduced to avoid the Girard paradox and maintain logical consistency, the multiple universes present in a type theory influence its logical power and behaviour, depending on the new definition of the Sort and Pi typing rules.

Impredicativity, cumulativity The universe hierarchy of CC_ω is said to be *predicative* because of the Pi typing rule. This rule states that a dependent product lives in a universe that is larger than those of its domain and codomain. This means that each quantification in a term increases its universe level, depending on the universe of the type over which we are quantifying. The Coq proof assistant implements this hierarchy by representing $\Box_i$ with the term `Type@{i}`,[16] but it also includes other universes, such as a $\mathbb{P}$ universe represented by the term `Prop` called the universe of *propositions*. This universe has the particularity of being *impredicative*, *i. e.*, when the codomain of a dependent product is $\mathbb{P}$, then the Pi rule puts the entire dependent product in $\mathbb{P}$ regardless of the universe of the

16: In many cases, we leave the universe implicit and write `Type`. This feature called *typical ambiguity* is presented with the other inference features of Coq, in section 3.1.

domain. As the behaviour of this universe in typing is different from that of predicative universes, it is considered present only in the first prototype developed during this thesis.

The Sort rule defines the inclusion policy between universes. Coq then defines an additional version of the rule to define the behaviour of $\mathbb{P}$, but also another rule known as *cumulativity*, allowing a universe in the hierarchy to be included in any universe above it:

$$\frac{i < j}{\Gamma \vdash \square_i : \square_j}\ (\text{Cumul})$$

This generalisation of the Sort rule presented in Figure 2.3 makes the calculus more flexible, but it also makes is harder to reason about universe levels, because of the many additional cases it allows.

Universe polymorphism Typing rules mentioning universe levels impose constraints on these universes. Historically, in Coq, each occurrence of a universe is associated with a global variable representing its level. These constraints on the order between universes that appear when typechecking terms are then stored in a global constraint graph that must always be valid, *i. e.*, acyclic. There is no attempt to determine an exact integer to be assigned to each universe level, but the validity of the universe constraint graph guarantees the existence of a solution, which is sufficient to maintain logical consistency in the current development. For example, here is the definition of a type and a constructor for that type:

$$\begin{aligned} \text{Box} &: \square_\alpha \to \square_\beta \\ \text{box} &: \Pi A : \square_\gamma .\, A \to \text{Box}\ A \end{aligned}$$

Levels α, β and γ are then constrained by all the uses of these two constants.[17] In some cases, this approach can cause typing errors. For example, identity can conceptually be applied to any term:

17: The very definition of box contains an occurrence of Box, adding the following constraint by rules App and Cumul:

$$\gamma < \alpha$$

$$\begin{aligned} &\text{id} : \Pi A : \square_\delta .\, A \to A \\ &\text{id}\ \mathbb{N}\ 0 : \mathbb{N} \\ &\text{id}\ (\mathbb{N} \to \mathbb{N})\ (\lambda n : \mathbb{N}.\, n) : \mathbb{N} \to \mathbb{N} \end{aligned}$$

However, in this variant of type theory, applying it to itself is impossible. Indeed, if we apply identity to itself, the first parameter of the application is the type parameter, which must live in $\square_\delta$. This parameter is the type of the identity:

$$\Pi A : \square_\delta .\, A \to A$$

Its domain is $\square_\delta$ of type $\square_{\delta+1}$ and its codomain is $A \to A$ of type $\square_\delta$. By the Pi rule, the type of identity therefore lives in universe $\square_{\delta+1}$, which is higher than $\square_\delta$. Consequently, it is impossible to apply identity to itself.

In terms that do not exploit a particular level of universe but the relation between different universes within the same term, one way to obtain the desired behaviour is to consider the universes as bound variables. By doing this, we define not a single constant but a family of constants indexed by a list of bound universes that we then call a *universe instance*. Identity becomes the following term, indexed by a universe variable i:

$$\text{id}_i : \Pi A : \square_i .\, A \to A$$

The well-typed version of the application of identity to itself is then $\mathrm{id}_{i+1}\ \mathrm{id}_i$. This feature is called *universe polymorphism* [27], and identity is said to be a polymorphic constant over universes, *i. e.*, a *universe-polymorphic* constant. Each occurrence of a universe-polymorphic constant is then constrained independently in the universe constraint graph, rather than globally in the monomorphic case.

[27]: Sozeau *et al.* (2014), "Universe polymorphism in Coq"

Nevertheless, the practical implementation of universe polymorphism raises non-trivial algorithmic questions. The answer chosen by Coq is therefore not irrevocable and continues to evolve in current versions of the software.

2.2.2 Inductive types

High-level programming languages allow developers to define custom data types. In statically typed functional languages, these are traditionally Algebraic Data Types (ADTs), or even Generalised Algebraic Data Types (GADTs). The Coq proof assistant features a version of ADTs, extended to dependent types and logically consistent, called *inductive types* [28].

[28]: Paulin-Mohring (1996), "Définitions Inductives en Théorie des Types"

Definition and pattern matching For example, consider the definition in Coq of the inductive type `nat`, corresponding to the theoretical type of natural numbers $\mathbb{N}$ taken as an example more than once previously:[18]

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

18: This type would be declared in the following way, in Haskell and in OCaml:

```
data Nat = O | S Nat
type nat = O | S of nat
```

This definition includes the definition of the `nat` type constructor, both constructors `O` and `S`, as well as the registration of these constants in the typing and reduction rules of Coq specific to inductive types.[19] The definition above states that a value of type `nat` is constructed from one of its two constructors. It is therefore possible to perform a case analysis on any value of type `nat` to find out its head constructor, by doing a *pattern matching*:

19: The definition of an inductive type also includes the definition of the induction principles on this type, presented at the end of the chapter.

```
Fixpoint add (n1 n2 : nat) : nat :=
  match n2 with
  | O ⇒ n1
  | S n ⇒ S (add n2 n)
  end.
```

According to the Curry-Howard correspondence, pattern matching also represents the case analysis of proof theory. Consequently, the typing rule of pattern matching must check its *exhaustivity*, in order to make it impossible to forget a case.[20] Coq therefore prohibits the definition of partial functions, whereas a functional language further away from a logical interpretation would simply give a warning from the compiler in such cases. The `Fixpoint` keyword indicates that the definition is recursive. The underlying λ-term then uses a `fix` recursion operator whose typing rule checks that recursive calls are *structurally decreasing*, *i. e.*, the function is called only on subterms of the argument of the current call. This allows recursion while guaranteeing that all programs terminate, as non-termination causes logical inconsistency.

20: It is even possible to declare a type without constructors. As there is no way to build a value in this type constructively, it encodes the concept of falsity.

These features are crucial in justifying the extra confidence that can be placed in a Coq proof over a paper proof. However, the terms considered in the following parts of this thesis do not contain pattern matching, which is why the previous

section only deals with the core of the theory, CC_ω, and the rest of this thesis only makes passing mention of pattern matching.

Data structures Inductive types can be used to define many different data structures, including different encodings of the same mathematical concepts. For example, here is an example of binary encoding of natural numbers:

```
Inductive bin_nat : Type :=
  | b0 : bin_nat
  | bpos : positive → bin_nat.

Inductive positive : Type :=
  | pH : positive
  | pI : positive → positive
  | p0 : positive → positive.
```

A binary number is either 0 represented by `b0`, or a non-negative number encoded starting from the least significant bit, `p0` representing 0, `pI` representing 1, and `pH` being the first 1 at the head of the number, with which all numbers start since the case of `b0` has been eliminated. Here are some binary numbers and their representation in this format:

1	$(0b1)$	$\mapsto$	`bpos pH`
2	$(0b10)$	$\mapsto$	`bpos (p0 pH)`
6	$(0b110)$	$\mapsto$	`bpos (p0 (pI pH))`

It is possible to use both types `nat` and `bin_nat` for representing natural numbers. This diversity offered by the proof assistant is at the heart of the questions addressed during this thesis.

One can also define data structures that make full use of polymorphism and dependent types. For example, here are definitions of linked lists and a function that computes the length of a list, in Coq:

```
Inductive list (A : Type) : Type :=
  | nil : list A
  | cons : A → list A → list A.

Fixpoint length (A : Type) (l : list A) : nat :=
  match l with
  | nil _ ⇒ 0
  | cons _ _ l' ⇒ S (length l')
  end.
```

The `list` type takes an argument `A` in its definition, so it is a polymorphic type: one can build values of type `list nat`, `list (list nat)`, `list (nat → nat)`, *etc.* The fact that the argument is located before the announcement of the universe in which the inductive type lives in the head of the definition — character `:` — makes it a *parameter*, *i. e.*, all the constructors listed below invariably build values of type `list A`. The types of the constructors therefore contain a quantification over the parameter, left implicit in the definition of the inductive type but clearly visible when it is used: the branches of the pattern matching in the definition of `length` include this first argument, in an ignored form `_` because it does not contribute to the function's output value. Indeed, the real types of constructors for `list` are the following:

```
nil : forall (A : Type), list A
cons : forall (A : Type), A → list A → list A
```

Finally, dependent inductive types can be defined, such as the type of vectors of fixed size:

```
Inductive vec (A : Type) : nat → Type :=
  | vnil : vec A 0
  | vcons : forall (n : nat), A → vec A n → vec A (S n).
```

Here, the second argument to give to `vec` in order to obtain a type is a value of type `nat` corresponding to the length of the vector. As this value is located in the type of `vec`, after character `:`, it can change from one constructor to another. It is therefore an *index* and not a parameter. Pattern matching on a value `v` of type `vec A n` is then a dependent pattern matching, since it performs both a case analysis on the head constructor of `v` and on the head constructor of the value `n` located in the *type* of `v`. Furthermore, dependent pattern matching allows eliminating impossible cases. To illustrate this point, here is the definition of a dependent function that reads the head value of a vector:

```
Definition head (A : Type) (n : nat) (v : vec A (S n)) : A :=
  match v in vec _ m
  return match m with 0 ⇒ nat | S _ ⇒ A end
  with
  | vnil _ ⇒ 0
  | vcons _ _ a _ ⇒ a
  end.
```

As this function makes no sense in the case of an empty vector, it is only defined on vectors of type `vec A (S n)` for a given `n`. The pattern matching in Coq makes it possible to specify the type of the inspected value by means of an `in` clause and the type of the value returned in the branches by means of a `return` clause. In the case of dependent pattern matching, one can make the type of the return value depend on the type of the inspected value. In the case of `head`, we bind the size of the vector to a variable `m` and we state that the pattern matching will return a value in a type that depends on the value of `m`. If `m` is `0`, a natural number will be returned, otherwise it will be a value of type `A`. The pattern matching branches reflect this case analysis in the return type, because the branch of the empty vector returns `0` and the other branch returns the head value `a` of the vector, which is of type `A`. As the value `v` is never an empty vector, the pattern matching will never actually take the first branch, but it must be defined for exhaustiveness reasons. As the case of the empty vector is meaningless but still needs to be defined, we use an arbitrary return type for this case in the `return` clause, but trivial to inhabit, so that the associated branch is easy to fill. In this case, we choose `nat` and give the inhabitant `0`. Thanks to the expressive power of dependent types, this function is well typed and the type system guarantees that it can never be applied to empty vectors.

Encoding additional constructs Inductive types in Coq can also be used to implement other constructions frequently used in type theory. Here are examples of definitions for the product type $\times$ and the sum type $+$:

```
Inductive product (A B : Type) : Type :=
  | pair : A → B → product A B.
```

```
Inductive sum (A B : Type) : Type :=
  | inl : A → sum A B
  | inr : B → sum A B.
```

A pair of type $A \times B$ is built using two values of type A and B, and a sum of type $A + B$ has two cases, either a value of type A, or a value of type B.

Dependent inductive types can also be used to encode more advanced constructions. A dependent pair $\Sigma x : A. B$ is a pair $(x\,; b)$ in which x is a value of type A and b is a value in a type B that is *allowed to depend on* x. Such a construction can be taken as one of the base constructions of the language under study, with the same status as the Π-type or the abstraction, or it can be encoded using terms of the language. In Coq, a dependent pair can be represented by the following type:

```
Inductive sigma (A : Type) (B : A → Type) : Type :=
  | dpair : forall (a : A), B a → sigma A B.
```

A pair `dpair a b` contains a value `a` of type `A` and a value `b` of a type `B a` that depends on `a`.

A second advanced construction that can be defined using dependent inductive types is equality, in a version called *propositional equality*. An equality $x = y$ can be encoded as a term `eq A x y`, where `A` represents the type of `x` and `y`:

```
Inductive eq (A : Type) (a : A) : A → Prop :=
  | refl : eq A a a.
```

Equality therefore has only one constructor describing the fact that the only possible value equal to a value `a` is `a` itself. Thanks to dependent pattern matching, in a proof containing a hypothesis `e` of type `eq A x y`, when typing allows it, performing a case analysis on `e` exposes the only possible case, *i. e.*, that `y` is exactly the term `x`, in the sense that these terms become *convertible*. This case analysis then allows replacing all the occurrences of `y` with `x` and the occurrences of `e` with `refl A x`. Conversely, when a property of type `eq A x x` must be proved, it is sufficient to supply the `refl A x` term to complete the proof. Since conversion includes the reduction rules of the calculus, the `refl` constructor can be used even when the equality has two syntactically different but convertible terms. For example, we can use `refl` to prove the following property, representing $1 + 1 = 2$:

```
eq nat (add (S O) (S O)) (S (S O))
```

Records Thanks to inductive types, we can also encode *records*, present in many programming languages to structure data. Here is the Coq definition for a record type representing non-negative coordinates in two dimensions:

```
Record Coord := { x : nat; y : nat }.
```

A record is then defined by giving a value to each field:

```
Definition coord_origin := {| x := 0 ; y := 0 |}.
```

The definition of a record type is equivalent to the definition of an inductive type with a single constructor taking two arguments, one for each field of the record, and two projection functions to extract each field from an inhabitant of the freshly created type.

```
Inductive Coord : Type := BuildCoord : nat → nat → Coord.
Definition x (c : Coord) : nat := match c with BuildCoord x _ ⇒ x end.
Definition y (c : Coord) : nat := match c with BuildCoord _ y ⇒ y end.
```

In non-dependent programming languages, a record can be encoded by a tuple, since this is just a construction allowing several values to be grouped together in the same term, by naming them using projections. In a dependent context such as that of Coq, an analogous encoding of records requires Σ-types. Since record types are in fact inductive types, they can naturally be polymorphic or dependent, and each of their fields is allowed to depend on the previous fields. This functionality is exploited in the definition of mathematical structures in the proof assistant. For example, a monoid is a type M with an associative operation $\diamond : M \to M \to M$ and an element $m_0 : M$ neutral for $\diamond$. In Coq, this mathematical structure can be represented with a record type that, by virtue of the Curry-Howard correspondence, contains both data — the definitions of m_0 and $\diamond$ — and properties[21] — the proofs of associativity of $\diamond$ and neutrality of m_0 for $\diamond$:

21: Also called *laws*.

```
Record Monoid (M : Type) := {
  mzero : M;
  mconcat : M → M → M;
  mconcat_assoc : forall (m1 m2 m3 : M),
    mconcat m1 (mconcat m2 m3) = mconcat (mconcat m1 m2) m3;
  mzero_neut_mconcat_l : forall (m : M), mconcat mzero m = m;
  mzero_neut_mconcat_r : forall (m : M), mconcat m mzero = m
}.
```

An instance of a mathematical structure on a given type is therefore an instance of the record type, so its data can be used in programs and its properties in proofs. This approach differs in particular from that chosen by Haskell, whose standard library defines several structures, including the monoid, but also the functor or the monad, only by their data [29]. As Haskell is not used to carry out proofs internally in the language, compliance with the various laws of mathematical structures depends on the user's discipline in declaring instances of these structures. It is then possible to define illicit instances that, despite their utility — these structures provide abstractions that are useful in programming —, make invalid any reasoning involving the laws of the structure carried out on a program that uses these instances. As the laws are directly in the record type, any instance of a structure defined in Coq is forced to abide by them, as its definition includes their proofs.

[29]: Wadler (1995), "Monads for functional programming"

From an inductive definition, Coq automatically generates and proves an *induction principle*, *i. e.*, a lemma that can be used in proofs to carry out reasoning by induction on a value of this inductive type. Here is the induction principle generated when defining the type nat:[22]

22: In reality, a separate induction principle is generated for each existing sort in the calculus, to cover all possible codomains for property P, but we are only interested in the version for Type in this thesis.

```
Lemma nat_rect : forall (P : nat → Type),
  P 0 → (forall (n : nat), P n → P (S n)) → forall (n : nat), P n.
```

Note that this induction scheme describes the traditional mathematical induction on natural numbers: if a property is true for 0 and is transmitted from any number to its successor, then it is true for all natural numbers. The induction scheme for lists is similar, with a hypothesis for the empty list and a hypothesis for the addition of a value at the head of the list. As the type of lists is polymorphic, so is the generated induction principle:

```
Lemma list_rect : forall (A : Type) (P : list A → Type),
  P (nil A) → (forall (a : A) (l : list A), P l → P (cons A a l)) →
    forall (l : list A), P l.
```

Well-formedness and induction schemes Apart from the presence of dependent types, the major difference between inductive types in Coq and algebraic data types available in other programming languages is that when they are declared, the proof assistant ensures that it is possible to reason inductively about these types. This means that not all inductive types can be defined in Coq. For instance, the following inductive type, whose values are constructed from a function producing values in this same type, is invalid in Coq:

```
Fail Inductive I : Type :=
  | K : (I → I) → I.
```

Indeed, the occurrences of `I` in the type of `K` are not all covariant, and the definition of this inductive would allow proving the following induction principle:

```
I_rect :
  forall (P : I → Type),
    (forall (f : I → I) (i : I), P (f i) → P (K f)) →
      forall i : I, P i
```

However, by instantiating `P` with `fun _ ⇒ False`, it becomes possible to prove $\bot$ and break logical consistency:

```
I_rect (fun _ ⇒ False) (fun _ _ F ⇒ F) (K (fun i ⇒ i)) : False
```

As a result, Coq has a *positivity checker*, *i. e.*, a criterion for well-formedness of inductive types, which ensures that logical consistency is maintained if all inductive types meet it.[23]

23: This criterion being correct but not complete, there are theoretically valid inductive types that are not accepted by the proof assistant.

Proof assistance 3

This section discusses the various features making Coq a genuine proof *assistant*. It is a partial overview covering the elements showing up in the rest of this thesis. Firstly, Coq offers inference features allowing the user to only partially write the terms and delegate the rest of the work to the proof assistant (§ 3.1). Secondly, the software features a proof mode with a dedicated language, so that it is possible to make progress in a proof step by step and view its current state at any time, without having to write the proof terms manually (§ 3.2). Thirdly, the proof assistant provides a means of easily exploiting equality and more generally equivalence relations between terms, so that they are interchangeable in a relatively transparent way (§ 3.3).

3.1 Inference 22
3.1.1 Unification 22
3.1.2 Inference and *ad hoc* polymorphism 25
3.2 Tactics and automation 26
3.2.1 The proof mode 26
3.2.2 Automated proof tactics 29
3.3 Rewriting and proof transfer . 31
3.3.1 Rewriting 31
3.3.2 Extension to equivalence . . . 32

3.1 Inference

Most statically typed languages come with inference features: Hindley-Milner-style languages [30, 31] leading the way, their theoretical basis being designed with the aim of *complete* inference, but also more common languages. For example, in modern versions of C++, the use of the `auto` keyword allows letting the compiler attempt to find the type of a variable or the return type of a function. In the case of Coq, the programming language is extremely complex, so inference systems need to be more advanced. In this section, we present various features that allow the user to *implicitly* leave *holes* in terms when some information can be inferred from the elements they supply.

[30]: Hindley (1969), "The principal type-scheme of an object in combinatory logic"

[31]: Milner (1978), "A theory of type polymorphism in programming"

3.1.1 Unification

The complexity of the language of Coq brings a degree of verbosity in terms, that can sometimes amount to obfuscation. For example, here is a Coq term corresponding to the concatenation of two lists of integers $[1] \diamond [2, 3]$, in its raw version presented in the previous chapter:

```
mconcat (list nat) (list_Monoid nat)
  (cons nat (S O) (nil nat))
  (cons nat (S (S O)) (cons nat (S (S (S O))) (nil nat)))
```

This assumes the existence of a monoid instance for lists in the context:

```
list_Monoid : forall (A : Type), Monoid (list A)
```

Unification variables The repetition of type `nat` $n+1$ times to build a list of n values seems redundant to the user, but it is necessary for the resulting term to be well typed. The information is indeed superfluous, since it suffices, for example, to systematically give as first argument for `cons` the type of the value added at the top of the list, *i.e.*, the next argument. The instantiation of `list_Monoid` to `nat` can also be inferred from the context. The actual implementation of Coq has a special term constructor called *unification variable* designed specifically to be able to leave holes in terms. The example above can then be written as follows:

```
mconcat _ (list_Monoid _)
  (cons _ (S O) (nil _))
  (cons _ (S (S O)) (cons _ (S (S (S O))) (nil _)))
```

For each hole left in the term, Coq creates a unification variable. The typechecker must then solve all the unification problems and fill in the holes before accepting the term. For example, the unification problem for the first list is the following:

```
cons ?T (S O) (nil ?T) : list ?T
```

Thanks to the type of `S O`, we can infer that `?T` is type `nat` and know that the rest of the list must have type `list nat`, which triggers the next unification problem `nil ?T : list nat`. At the end of the inference process, all the annotations have been added to the term without any help from the user.

Implicit arguments The inference features of Coq allow going beyond the explicit creation of unification variables by leaving them *implicit*. Indeed, as the first argument of constructors of type `list`, the type parameter, is inferrable from the context, it can be made implicit by using special syntax in the declarations, or by executing dedicated Coq commands. In the case of an inductive type, it is not necessarily desirable to make the parameter implicit everywhere, as this would allow declaring values of type `list` without mentioning this parameter, which could lead to confusion. In this case, the following commands are used for constructors:

```
Arguments nil {_}.
Arguments cons {_}.
```

Thus, the constructor of type `list` always needs an explicit parameter, but a value of type `list nat` can be constructed just by writing `nil`. One can also declare the parameter of `list_Monoid` and the parameter of `mconcat` as implicit. The above example then becomes the following one:

```
mconcat list_Monoid
  (cons (S O) nil)
  (cons (S (S O)) (cons (S (S (S O))) nil))
```

In the definition of a polymorphic function, one can use a syntax that indicates that the polymorphic parameter is implicit. For example, the head of the declaration of the `length` function defined in the previous chapter becomes the following, where the braces declare an argument as implicit:

```
Fixpoint length {A : Type} (l : list A) : nat.
```

We then write `length l` as in a non-dependent language.

In the context of dependent types, it is even possible for a value to be inferrable without being a type. For example, the `head` function retrieving the head of a vector takes as argument an integer that appears in the type of the next argument, the vector whose head we want to extract. This integer can therefore be declared as implicit and inferred for each occurrence of `head`. Thus, one can write `head (vcons 4 vnil)` without giving any vector size, neither in constructors of type `vec` nor in the `head` function.

Notations In order to make terms even more readable, Coq offers a *notation* system adding cases to the parser to associate a particular syntax to some terms. These notations can be infix and the user can define priorities between them. In particular, the standard library defines notations for arithmetic and lists, in order to obtain terms in a syntax that is very close to that of other programming languages. Notations `+` and `*` are associated with addition and multiplication operations on natural numbers, and numerical constants can be written naturally in base 10. A recursive notation is associated to lists so that they can be expressed in the syntax of OCaml. Thus, the example in this subsection becomes:

```
mconcat list_Monoid [1] [2; 3]
```

In the rest of this document, we will use classic Coq notations, such as `=` for equality, `*` for products, *etc.*

Coercions Classically, statically typed functional programming forbids implicit casts between types by default. Then, a boolean value cannot be used in a position where the typechecker expects an integer. However, the proof assistant gives the user the option of declaring a function as a *coercion*, *i. e.*, an implicit cast. In this case, a function `nat_of_bool` of type `bool → nat` that would associate `true` with `1` and `false` with `0` can be declared as a coercion using the following command:

```
Coercion nat_of_bool : bool >-> nat.
```

Thus, any typing problem `b : nat` where `b` is a value in `bool` becomes a unification problem `?f b : nat` where `?f` is a coercion. Coq then attempts to build this coercion, possibly by transitivity. This powerful mechanism provides additional flexibility in the syntax but increases the risk of having a term validated by the typechecker in a case where the user would expect a typing error, with the typechecker inserting a meaningless coercion. For example, we can define a coercion that encodes a natural number with a pair of integers defining its quotient and remainder in a Euclidean division by a given constant. In such a case, if the user *mistakenly* enters a boolean in a context where a pair of integers is expected, Coq may compose the coercions and validate the term, possibly causing errors that are difficult to trace later on, whereas a typing error would directly show the problem.

Typical ambiguity Most examples of Coq terms in this thesis do not specify universe levels. Indeed, Coq has a feature called *typical ambiguity* that allows inferring them automatically. The associated universe constraints are then checked before concluding that the term is well typed. In many cases, this inference is sufficient and gives the user the illusion that the entire hierarchy of predicative universes is a single universe, which is more natural and makes it easier to learn Coq, although theoretically incorrect. On the other hand, in the context of universe polymorphism, this inference is imperfect and sometimes has to be disabled and annotations made by hand. Typically, when development includes circular reasoning in which a term contains itself, it may be necessary to enable universe polymorphism. In order to be sure that we define terms at the right universe levels, we can impose a strict universe instance and reduce the scope of universe inference. In this thesis, the implementation of the Trakt prototype uses typical ambiguity, but the implementation of the Trocq prototype specifies many universes manually.

3.1.2 Inference and *ad hoc* polymorphism

The various inference features presented here take their information from the context of the unification problem, but it is also possible to retrieve them from a meta-level database fed by the user. In particular, this allows implementing *ad hoc* polymorphism in some programming languages, *i. e.*, the definition of values, not by quantifying universally over a type as it is the case for general polymorphism presented earlier, but for a finite subset of types. In this way, a generic operation can be used on any inhabitant of a type contained in this subset, and the membership witness is inferred by the type system.

Typeclasses The most common way of implementing *ad hoc* polymorphism is to use *typeclasses*. These are structures whose instances can be registered as canonical inhabitants of their type. Similarly to HASKELL [32] or SCALA [33], COQ has this feature [34]. It can be enabled by using the `Class` keyword at the head of the structure definition. If the monoid is defined as a typeclass, then instances can be registered by declaring them with the `Instance` keyword. One can then define notations that introduce inference problems that will be resolved automatically using the database of instances of the typeclass:

[32]: HALL *et al.* (1996), "Type classes in Haskell"

[33]: OLIVEIRA *et al.* (2010), "Type classes as objects and implicits"

[34]: SOZEAU *et al.* (2008), "First-class type classes"

```
Notation "x <> y" := (@mconcat _ _ x y).
```

Here, the first hole is the type declared as a monoid, and the second one is the instance of type `Monoid _`.

It is also possible to declare type-level *functions* producing canonical typeclass instances from other instances. For example, if two types A and B are monoids, then the product type $A \times B$ is also one. The head of the definition in COQ is the following:

```
Instance product_Monoid {A B : Type}
  `{M_A : Monoid A} `{M_B : Monoid B} : Monoid (A * B).
```

This inference method makes it possible to make reliable use of genuine genericity in the syntax. For example, thanks to this notation and the declaration of `list_Monoid` as the canonical function for making instances of monoids on lists, the example from the previous subsection is written in the following way, which is exactly the same syntax as the mathematical notation:

```
[1] <> [2; 3]
```

Some difficulties remain, such as the analogous problem of diamond inheritance in object-oriented programming, where an instance can be inferred by several paths, or the inference of complex terms in the context of dependent types, where an instance can be indexed by something else than a type.

Canonical structures Mathematical structures are defined as a type called *carrier type*, provided with particular data — values and/or operations — linked to this type, as well as laws governing these data. One way of encoding these structures in a programming language is to use a record type containing the operations and laws, and to use the carrier type as a parameter of this record to make it polymorphic, as it is the case in the definition of `Monoid` in the previous chapter. To increase automation, we can then turn this record type into a typeclass.

Another solution is to put the carrier type directly in the structure, as the first field. As records are dependent, the definition of the other fields is not impacted. This is

the solution chosen by the MathComp library [35], as shown by the eqType structure representing types equipped with a decidable equality. Here is a declaration similar to the one made in the library:[1]

```
Record eqType : Type := {
  carrier : Type;
  eq_op : carrier → carrier → bool;
  eq_op_equality : forall (x y : carrier), x = y ↔ eq_op x y = true
}
```

In this case, we can define the first projection as a coercion, thus making it possible to write proofs looking like they quantify over an instance of the structure, but in reality quantify over its carrier type. In order to recover the features of typeclasses, Coq proposes a way of declaring *canonical structures*[2] [37] to automatically solve the following unification problem, with T a concrete type and ?E a unification variable that Coq must fill with the correct structure:

$$T \equiv \text{carrier } ?E$$

Generic notations can then be defined, such as == for eq_op, which can be used for any instance of eqType. For example, in the following lemma, values T_1 and T_2 appearing in the type of u and v hide implicit coercions towards their carrier type, and we use the generic notation for eq_op on these values:[3]

```
Lemma pair_eq1 : forall (T₁ T₂ : eqType) (u v : T₁ * T₂),
  u == v → u.1 == v.1.
```

Here, u.1 == v.1 is actually eq_op T_1 u.1 v.1, the canonical structure inference function having filled in the term automatically.

[35]: Mahboubi *et al.* (2021), *Mathematical Components*

1: The actual declaration exposes a degree of complexity that does not need to be presented here. In particular, it calls on another library, Hierarchy Builder [36], that automates the creation of nested structures, not covered in this thesis.

2: This is also the name of the feature.

[37]: Mahboubi *et al.* (2013), "Canonical structures for the working Coq user"

3: Here, .1 is a notation for a generic function defined on products allowing extraction of the first component.

3.2 Tactics and automation

The inference presented in the previous section makes the language of Coq more flexible and therefore easier for the user to write. What makes Coq a true proof assistant, however, is the exploitation of inference in a *proof mode* displaying to the user the current state of the proof until it is completed, making it easier to identify the next action to perform. These actions are represented by specific proof mode commands called *tactics*. In this way, the proof makes progress step by step without the user having to write the proof terms by hand, and the final proof comes in the form of a script in this tactic language called Ltac [38], which is more readable and closer to a paper proof. Some tactics do more than just performing a proof step, and build entire proofs of statements contained in a specific theory that they know how to process, thus offering true proof automation. This section presents the proof mode and these advanced tactics.

[38]: Delahaye (2000), "A tactic language for the system Coq"

3.2.1 The proof mode

When making a definition, after writing the head of the definition containing the name and type of the term to be declared, we can give this term in raw form as we did earlier in the examples, or enter the *proof mode* with the Proof keyword. The type of the term we declare then becomes a type to be inhabited, called *goal*. The proof is performed by executing a series of *tactics*, each of which generates a proof term. Each proof term is given to Coq and advances the proof more or less

according to its type. If its type is not convertible to the goal, then the tactic fails and the user must change strategy or cancel the proof. If its type is convertible to the goal, then we inspect it to see if it contains any unification variables. If there are no unification variables, the proof is complete. Otherwise, each unification variable is a hole that still needs to be filled in order to complete the proof: Coq then creates one subgoal per unification variable and asks the user to prove all the subgoals. The primitive tactic that exactly reflects this behaviour is `refine`, but this presentation of the proof mode focuses on tactics that are closer to logical reasoning.

Let us illustrate the proof mode by analysing the proof of the following lemma in Coq:

```
Theorem length_append {A : Type} : forall (l1 l2 : list A),
  length (append l1 l2) = length l1 + length l2.
```

First of all, here is a definition of the concatenation function over lists:

```
Fixpoint append {A : Type} (l1 l2 : list A) : list A :=
  match l1 with
  | nil ⇒ l2
  | cons a l ⇒ cons a (append l l2)
  end.
```

The initial state of the proof is the following one, represented by hypotheses at the top and the goal to prove at the bottom:

```
A : Type
--------------------------------------------------------------------
forall (l1 l2 : list A), length (append l1 l2) = length l1 + length l2
```

Initially, we can use the `intros`[4] tactic to come under the quantifiers and introduce into the context two lists l_1 and l_2.

4: `intros l1 l2.`

```
A : Type
l1, l2 : list A
--------------------------------------------
length (append l1 l2) = length l1 + length l2
```

Indeed, to prove that a property is valid on any pair of lists, we can name these values and prove the specialised property on these values, *i. e.*, by instantiating the quantifiers.

Then, as we have to prove a property on any list, we can make a case analysis on l_1 with the `destruct`[5] tactic. Coq therefore divides the proof into two sub-cases, one in which the list has been replaced with an empty list, and the other in which the list has been replaced with a list `a :: l`.

5: `destruct l1 as [a l].`

```
A : Type
l2 : list A
------------------------------------------------ (G1)
length (append nil l2) = length nil + length l2

A : Type
a : A
l : list A
---------------------------------------------------------- (G2)
length (append (a :: l) l2) = length (a :: l) + length l2
```

In the first subgoal G1, since in the case of an empty list, the length is zero according to the definition of `length`, addition is equal to the second value and

concatenation is the second list according to the definition of `append`, the goal is convertible to `length l₂ = length l₂`. Therefore, it is sufficient to apply the `reflexivity` tactic, that ends the proof by telling Coq it is a trivial equality. This tactic call is equivalent to applying the `refl` constructor of equality with the `exact`[6] tactic. This tactic allows applying a proof term expressed in the language of Coq. Thanks to the `apply`[7] tactic, it is also possible to tell Coq that it is sufficient to apply `refl` to an argument, and let the inference take care of finding this argument. This makes the final proof script more readable. The `eapply` variant[8] adds flexibility by allowing Coq to add unification variables in the event that the inference does not find the right argument, but it is not necessary here.

6: `exact (refl (length l2)).`

7: `apply refl.`

8: And in general, tactics prefixed by an `e`.

To prove the subgoal G2, which then becomes the only remaining goal, we can perform a reduction step with the `simpl`[9] tactic to see that the presence of `a` corresponds to an incrementation of the lengths on both sides of the equality.

9: `simpl.`

```
A : Type
a : A
l, l2 : list A
-----------------------------------------------
S (length (append l l2)) = S (length l + length l2)
```

We can then see that the goal is impossible to prove as it stands. Except for the addition of `a : A` to the context and `S` on both sides of the equality — that can be deleted with the `f_equal` tactic —, the proof is in the same state as the initial one. This loop is symptomatic of the use of a too weak reasoning. A case analysis is not always sufficient to prove a property on any value of a type. Here, we need to reason by induction on the list instead of a simple case analysis. The first case G1 does not change, as it is the base case of the induction, but the second one G2 becomes an inductive case, with an induction hypothesis in its context. We can then use the `induction`[10] tactic instead of `destruct`.

10: `induction l1 as [|a l IHl].`

```
A : Type
a : A
l, l2 : list A
IHl : length (append l l2) = length l + length l2
-----------------------------------------------
   length (append l l2) = length l + length l2
```

Thanks to the induction hypothesis, we have exactly the element we need to conclude the proof. The final proof script is the following:

```
Theorem length_append {A : Type} : forall (l1 l2 : list A),
  length (append l1 l2) = length l1 + length l2.
Proof.
  intros l1 l2.
  induction l1 as [|a l IHl].
  - reflexivity.
  - simpl. f_equal. exact IHl.
Qed.
```

The subgoals created by the `induction` tactic are presented in the form of a bullet point list, for greater readability but also to signal to Coq that we are first focusing on the first one, then the second one. The calls to `simpl` can be left in the script if the reduction step contributes to a proof that is easier to understand for a human who would execute it step by step, but these calls are not strictly necessary since the typing of Coq includes conversion. The initial step introducing elements into the context can also be done before the proof, in the head of the definition:

```
Theorem length_append {A : Type} (l1 l2 : list A) :
  length (append l1 l2) = length l1 + length l2.
```

Finally, the proof is concluded with `Qed`, a line which when executed sends the global proof term to the kernel for a second check before registering the term in Coq. There is a subtlety in the closure of the proof mode: the `Qed` keyword makes the proof *opaque*. This means that after the declaration, the type of the proven statement is considered to be inhabited, and the witness is the declared term, but its *definition*, *i. e.*, the proof term, is in fact inaccessible, as if it was forgotten by the proof assistant. This can be useful to distinguish computational proofs from purely logical proofs, where one does not want to introduce lengthy terms in the goal because of an ambitious reduction step. If we wish to expose the proof term, we can make it *transparent* by closing the proof mode with the `Defined` keyword instead.

In most cases, declarations with a computational purpose are done by programming the term, because this is a more natural way to go, and proofs are made in the proof mode for the same reason. However, the boundary between both is not clear-cut. With dependent types, a proof can depend on data and data can depend on a proof. The best way to make the declaration is therefore not always obvious. For maximum control over the generated proof term, it may be interesting or even necessary[11] to manually supply the proof terms, at least partially using the `exact` tactic in proof mode to supply raw subterms and select the parts we want to delegate to Coq's inference.

11: It is notably the case for some proofs in the second prototype developed in this thesis, Trocq, presented in § III.

3.2.2 Automated proof tactics

The proof mode provides the user with a level of abstraction over Coq's λ-calculus in the creation of proofs, but it can also be used to automate them. Indeed, the tactics receive the current state of the proof and return a term justifying the transition to a new proof state, which is the final state if the term is sufficient to complete the proof. The tactics presented earlier are used to perform atomic proof steps, but nothing prevents the development of more advanced tactics, able to reduce proofs to a single line in the proof script.

Some tactics perform proof search by selecting lemmas in the context or in dedicated databases. This is the case for the `auto` tactic in the standard library, but also for much more complex tools such as *hammers*, inspired by the Sledgehammer [39] plugin for the Isabelle/HOL [8] proof assistant. Hammers first filter the global context to select the most relevant reachable lemmas for the proof to perform. Then, they send this context and the goal to be proved to automated theorem provers, and determine the minimal sub-context needed to make the proof. This minimal context is then given to a reconstruction procedure that performs the proof once again, this time within the proof assistant. The representative of this family of tools in the Coq ecosystem is the CoqHammer [40] project.

[39]: Böhme *et al.* (2010), "Sledgehammer: judgement day"

[8]: Nipkow *et al.* (2002), *Isabelle/HOL: a proof assistant for higher-order logic*

[40]: Czajka *et al.* (2018), "Hammer for Coq: Automation for dependent type theory"

Other tactics are implementations of *decision procedures*, algorithms able to automatically prove statements in a given theory. For example, the `lia` [17] tactic in the standard library is the implementation of a decision procedure for integer linear arithmetic. It allows proving any statement belonging to this theory in a single tactic call. Another example is the SMTCoq [41] project, connecting Coq to SMT solvers. The goal to prove is first encoded into the SMT-LIB language [42], a standard input format for this family of solvers, then the problem is given to an SMT solver instrumented to provide a trace of its execution giving hints as to how

[17]: Besson (2006), "Fast Reflexive Arithmetic Tactics the Linear Case and Beyond"

[41]: Ekici *et al.* (2017), "SMTCoq: A Plug-In for Integrating SMT Solvers into Coq"

[42]: Barrett *et al.* (2010), "The SMT-LIB Standard: Version 2.0"

to prove the goal. This trace is then fed to a reconstruction procedure in Coq, that makes the proof term for the original goal.

An example of a proof where an automation tactic can be used is the proof of an instance of the `Monoid` record for `nat`:

```
Definition nat_Monoid : Monoid nat.
```

The `econstructor` tactic allows the record constructor to be applied without naming it, by creating unification variables for all the fields. The inference then expects more information from the user to fill in these variables. Here, we want to fill in the fields manually one by one, so we turn these unification variables into subgoals using the `unshelve`[12] tactic. Initially, these goals mention the unification variables:

12: `unshelve econstructor.`

```
─────────────── (G1)
?mzero : nat

          ?mzero : nat
──────────────────────────────── (G2)
?mconcat : nat → nat → nat

              ?mzero : nat
              ?mconcat : nat → nat → nat
────────────────────────────────────────────────── (G3)
forall (a b c : nat),
  ?mconcat a (?mconcat b c) = ?mconcat (?mconcat a b) c

        ?mzero : nat
        ?mconcat : nat → nat → nat
──────────────────────────────────────── (G4)
forall (m : nat), ?mconcat ?mzero m = m

        ?mzero : nat
        ?mconcat : nat → nat → nat
──────────────────────────────────────── (G5)
forall (m : nat), ?mconcat m ?mzero = m
```

By focusing in turn on each subgoal in the order of the fields in the record, at each stage we prove a subgoal without unification variables. We choose to declare a monoid based on addition as the accumulator and zero as the neutral value. We therefore give these two values with the `exact`[13] tactic. The last three subgoals then become the following:

13: `exact 0.` / `exact add.`

```
──────────────────────────────────────────────── (G3)
forall (a b c : nat), a + (b + c) = (a + b) + c

──────────────────────────── (G4)
forall (m : nat), 0 + m = m

──────────────────────────── (G5)
forall (m : nat), m + 0 = m
```

The subgoal G3 can be proved by hand by induction on the integers, but if we notice that it falls into the theory of linear arithmetic, we can delegate the entire proof to the `lia` tactic. Finally, the last two subgoals are classical properties on natural integers, proved in lemmas of the standard library. In these cases, an automation tool such as `lia` is not necessary, but can simplify the proof script. Indeed, we can specify which goals we want to apply a tactic to, by prefixing the call with the numbers of the targeted goals, the prefix `all` applying the tactic to all the remaining goals. From the third subgoal onwards, the proof can be closed with a line by calling `lia` on all the remaining fields. The final proof script is therefore the following:

```
Definition nat_Monoid : Monoid nat.
Proof.
```

```
  unshelve econstructor.
  exact 0. exact add.
  all: lia.
Defined.
```

Note that the proof is concluded with `Defined` because we want to be able to extract `mzero` and `mconcat` from this term for computational purposes.

3.3 Rewriting and proof transfer

Another category of manipulations frequently carried out in mathematical proofs is *rewriting*, *i. e.*, substitution of values, mathematical structures, *etc.*, identified as similar. In the context of formal proofs, one must give this concept of similarity a formal definition, in order to justify the transition from the initial state to the substituted state in the kernel of the proof assistant. Equality is an example of relation that can be used to represent this notion of similarity. For example, when solving a system of equations, isolating a variable in one of the equations allows it to be expressed as a function of all the others and replaced with this new expression in all the other equations, thereby reducing the size of the problem.

In mathematical practice, equivalence relations that are more general than equality can be used to represent similarity between objects. This reasoning up to *equivalence* allows greater freedom in the proofs regarding the representation of mathematical objects. For instance, depending on the context and the proof to be carried out, it may be interesting to see a natural number through the prism of a unary encoding or a binary encoding, but the results obtained on unary integers can naturally be exploited in proofs on binary integers, and *vice versa*, as these two encodings are equivalent. In a Coq proof, the situation is not as simple, because the equivalence relations have to be formalised and the type system is less flexible. Indeed, in the context of dependent types, replacing one type with another is not trivial and can make a term ill typed. Making a proof performed with one encoding of a mathematical object available in the context of another encoding of this object is called *proof transfer*, and it is a non-trivial task although transparent on a paper proof. This section explains how rewriting works and discusses its extension to proof transfer in Coq.

3.3.1 Rewriting

The equality of Coq, represented by the inductive type `eq`, is based on the principle of *identity of indiscernibles*, attributed to Leibniz, according to which equality between two terms corresponds to the fact that these elements behave in the same way in all contexts. Formally, this property translates into the induction principle of equality: if a property is true for `x` and `x = y`, then the same property is true for `y`. Here is the type of the induction principle `eq_rect` in the standard library of Coq:[14]

14: In the specific case of `eq`, the induction principle used by default in Coq is non-dependent, *i. e.*, the property `P` does not also quantify over a proof of equality but only over the value of the index of type `A`.

```
eq_rect : forall (A : Type) (x : A) (P : A → Type),
  P x → forall (y : A), x = y → P y
```

This term can then be used to perform rewriting in a goal. Suppose that we wish to perform a rewrite between two values `y` and `x` of type `A` from an equality `e` of type `x = y`. To do this, it suffices to abstract the occurrences of `y` in the goal, to obtain a predicate of type `A → Type` that will be precisely the `P` argument of the

induction principle of equality. The term `eq_rect A x P ?p_x y e` is thus a proof of the goal, with a unification variable $?p_x$ corresponding to a new goal `P x`, *i. e.*, the same goal in which the selected occurrences of `y` have been replaced with `x`.

The tactic allowing rewriting using an instance of `eq_rect` as the underlying proof term is `rewrite`. It takes as parameters the direction of the rewrite — `x` to `y` or `y` to `x` —, the equality proof justifying it, as well as possible hints as to which occurrences we wish to rewrite and which we wish to leave untouched,[15] or the term we want to rewrite — by default, the goal. The tactic equivalent of the rewriting proof term is therefore `rewrite -> e` or `rewrite e`.

15: By default, the tactic tries to rewrite the value wherever it is possible to do so.

3.3.2 Extension to equivalence

Thanks to rewriting as presented above, we can exploit equalities in proofs. However, there are other situations in which a rewrite would be performed in a paper proof, not exploiting an equality but another equivalence relation instead. Indeed, Coq defines tactics `reflexivity`, `symmetry`, and `transitivity`, allowing respectively to apply `refl`, to flip an equality, and to cut a proof of equality in two, by going through an equality with a third term. Yet, the properties of reflexivity, symmetry, and transitivity are common to all equivalence relations. Consequently, Coq is equipped with a *generalised* rewriting [43] feature that extends these tactics as well as the `rewrite` tactic to *setoids* [44], *i. e.*, types with an equivalence relation. For example, consider the propositional equivalence ↔, *i. e.*, a two-way implication between two propositions in `Prop`. This relation is an equivalence in `Prop`, so it is possible to declare `Prop` as a setoid with this relation. Thanks to generalised rewriting, we can then prove `A <-> C` with the `transitivity` tactic and proofs of `A <-> B` and `B <-> C`, for a well-chosen proposition `B`.

[43]: Sozeau (2009), "A New Look at Generalized Rewriting in Type Theory"

[44]: Barthe *et al.* (2003), "Setoids in type theory"

Although extended to setoids, rewriting remains limited to equivalence relations, thus homogeneous relations. Rewriting in types is therefore a fragile action, because it can modify types in a statement but not the *values* that inhabit these types. Indeed, we can relate the unary and binary encodings of the natural numbers `nat` and `bin_nat`, declared in § 2.2.2, with an equivalence relation over types, but the relation cannot extend to values in these types that might be present in a goal. For example, consider the following goal:

```
forall (n : nat), 0 ≤ n
```

Here, we cannot rewrite `nat` into `bin_nat` without impacting the rest of the goal, because the occurrences of `n` change type in the process. In this case, we need a way to relate the order relation ≤ to an equivalent over `bin_nat`, and constants `0` and `b0` together, in order to *transfer* the whole goal from `nat` to `bin_nat`. This global operation is called *proof transfer* and can be used to transfer existing proofs or reformulate goals, along equivalence proofs.

The CoqEAL [45] project allows working with heterogeneous relations, that can be used to relate several representations of the same mathematical object, in particular a representation adapted to the proof of properties on the object and a more powerful representation for expressing programmes using the object. This is known as *refinement*. This plugin works on heterogeneous functional relations, but performs transfer only on closed terms without quantifiers. The rest of this thesis explores different options for proof transfer, first with the aim of preprocessing goals before running a proof automation tactic (§ II), then with the aim of

[45]: Cohen *et al.* (2013), "Refinements for Free!"

pushing further various recent proof transfer approaches based on *parametricity* [13], offering a maximum level of generality including dependent types but sometimes introducing axioms in a suboptimal way (§ III). The last part deals with implementation of such proof transfer techniques in Coq (§ IV).

[13]: Reynolds (1983), "Types, Abstraction and Parametric Polymorphism"

Meta-programming in Coq with Coq-Elpi

4

4.1 A logic meta-programming language for Coq 34
4.1.1 A logic programming legacy . . 34
4.1.2 Encoding of Coq terms 36
4.2 A toolbox 37
4.2.1 Databases 38
4.2.2 Creation of commands and tactics 39

The proof mode allows automating proofs by letting the user execute tactics to generate proof terms for them. Proof scripts are then written in the Ltac tactic language, that can be used to chain tactics calls and structure the proof. Tactics are written in a meta-language, the exact choice not being important for Coq since tactics can be seen as functions from one proof state to another. The only constraint on the tactic is that it must be able to provide a proof term that can be verified by the kernel afterwards.

Several meta-languages are available in Coq, each one with particular strengths: Ltac, OCaml, MetaCoq, Ltac 2, Coq-Elpi, *etc.* The Ltac language has functions to inspect the current proof state and create Coqterms, which makes it a meta-language that can be used to write tactics. The most primitive meta-language is OCaml, since it is the language used to implement Coq itself. Interactions with Coq are then made with the internal API of the proof mode and Coq terms are manipulated in their internal representation, which gives maximum freedom but exposes all the complexity to the meta-program. The MetaCoq project [46] allows manipulating Coq terms directly in Coq. It is therefore useful when one wishes to certify the meta-program, for example by proving its completeness. The aim of the Ltac 2 language [47] is to be a meta-language syntactically close to Ltac, while adding features expected in a modern language, such as a type system or a way to declare data structures. All the developments carried out during this thesis were done in the Coq-Elpi meta-language [16], presented in this chapter. We focus first on its features and then on the accompanying tools to interact with Coq.

[46]: Sozeau *et al.* (2020), "The MetaCoq Project"

[47]: Pédrot (2019), "Ltac2: tactical warfare"

[16]: Tassi (2018), "Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect)"

4.1 A logic meta-programming language for Coq

In reality, the Coq-Elpi plugin is an extension of a language called Elpi [48] to make it a complete meta-language for Coq. This section presents this language and the features that make it interesting in the context of meta-programming for Coq. We also explain how Coq terms are represented in Elpi.

[48]: Dunchev *et al.* (2015), "ELPI: Fast, Embeddable, λProlog Interpreter"

4.1.1 A logic programming legacy

The Elpi language belongs to a family of languages known as *logic programming languages*. This programming paradigm was developed in the second half of the 20th century with the advent of its most famous representative, Prolog [4]. The base object of such a language is the *predicate*, and programs have a logical interpretation in a subset of first-order logic.

[4]: Colmerauer *et al.* (1973), "Un système de communication homme-machine en français"

How a logical program works Rather than a series of instructions to be executed, a logical program is represented by a knowledge base and a query. The *knowledge base* is a list of declarations of facts — predicates that are unconditionally true — and rules — predicates that are true if a set of premises is true. Each declaration of a predicate is called an *instance* of that predicate, and can have arguments that are either *atoms* — base constant values of the language, such as

numbers or strings — or *variables*, *i. e.*, named locations that do not yet contain a concrete value.

The *query* is a list of predicate instances that we want to make true. An inference engine then explores the knowledge base to check whether a solution exists. For each predicate to be made true, the instances of this predicate in the knowledge base are browsed. For each instance, an attempt is made to syntactically unify the head of the instance — the fact or the conclusion of the rule — with the predicate instance to satisfy in the query. If unification succeeds, we replace this predicate in the query with the body of the instance — the conditions for the rule to be true —, which must also succeed for the query to have a solution. If unification fails, we move on to the next instance. If all the options are exhausted, then the request cannot be satisfied.

Logical languages are therefore said to be declarative, as they describe the problem and the characterisation of a solution rather than the procedure to compute this solution. More precisely, Elpi is an implementation of λProlog [49], an extension of Prolog adding λ-terms — predicates behave like functions —, quantifiers, and an implication operator.

[49]: Miller *et al.* (1987), *A logic programming approach to manipulating formulas and programs*

Let us illustrate how Elpi works with a simply-typed encoding of λ-calculus, presented in § 2.1.2.

```
kind lterm type.
type abs (lterm → lterm) → lterm.
type app lterm → lterm → lterm.
```

The code above declares a new type `lterm` representing the λ-terms, with two constructors, `abs` for the abstraction and `app` for the application. This is a HOAS encoding — *Higher Order Abstract Syntax* [50] —, *i. e.*, variables and functions in the object language are represented with Elpi variables and functions.

[50]: Pfenning *et al.* (1988), “Higher-Order Abstract Syntax”

We can define an Elpi type to represent the simple types of this λ-calculus:

```
kind ltype type.
type arrow ltype → ltype → ltype.
```

Similarly, the case of the type variable is represented by an Elpi variable. From these declarations, we can implement a typing predicate, taking a term from the calculus and returning a simple type.

```
pred type-of i:lterm, o:ltype.
type-of (app T1 T2) B :-
  type-of T1 (arrow A B),
  type-of T2 A.
type-of (abs F) (arrow A B) :-
  pi a\ type-of a A ⇒ type-of (F a) B.
```

The first line declares the type of the predicate, precising whether each argument is an input or an output. Each instance reflects a typing rule of the calculus, with the head of the instance representing the conclusion of the rule, and the body of the instance representing its premises. Note that the case of abstraction uses operators `pi` and ⇒, representing universal quantification and implication respectively. As the abstraction is represented by a meta-function, in order to inspect the body of this function, we need to provide it with an argument. The `pi` operator then locally introduces a variable `a`, called a *universal constant*. The term `F a` is then the body of the abstraction in which the bound variable is `a`. We check that this term has a type `B`, but this is only possible if we give a type to the

newly introduced variable a. The role of the implication is context extension and corresponds exactly to this situation: we add the hypothesis that a is of type A into the execution context of the predicate on the right hand side of the arrow. Thus, in the execution of `type-of (F a) B`, when we search for the type of a, the assumption we have provided will be an additional instance of the `type-of` predicate allowing us to give a type to this variable. The $\text{Var}_{\rightarrow}$ rule that involves the context is then implicit since it exploits the Elpi context.

Constraint handling rules A crucial feature of Elpi is the *Constraint Handling Rules* (CHR) language [51]. This feature allows making the control flow of Elpi programs more complex, by allowing requests to be *frozen* on a variable, *i. e.*, suspended until this variable takes a concrete value. A request R is frozen on a list of variables L by calling the following predicate:

[51]: Frühwirth (1994), "Constraint Handling Rules"

```
declare_constraint R L
```

In addition to freezing queries, it is possible to declare *rules* to detect the simultaneous presence of a set of frozen queries and execute Elpi code. For example, consider a binary predicate p_1 and a unary predicate p_2. We can declare a common constraint handling rule for these two predicates as follows:

```
constraint p₁ p₂ {
  rule (p₁ X Y) \ (p₂ 1) | Cond ⟺ Code.
}
```

The rule is triggered as soon as there is simultaneously in the set of frozen predicates an instance of p_1 applied to two arbitrary values and an instance of p_2 applied to the constant 1. When the rule is triggered, condition Cond is tested to see if the code of the rule — variable Code — must be executed. If the condition is true, the code is executed; otherwise, the other rules are inspected. Character \ is used to delete frozen requests once the rule has been executed: all predicates at the left of this character are kept, all those at the right are deleted. It is possible to write a rule that does not delete any identified predicates, or one that deletes them all.

4.1.2 Encoding of Coq terms

The Coq-Elpi plugin connects Elpi to Coq by defining internal predicates written in OCaml giving Elpi developers access to the proof assistant's API, in order to interact with Coq directly from the meta-language. When the body of a predicate is written in OCaml, it is necessary to define a way to connect OCaml values and Elpi values. Each manipulated data structure must then have a representation in both languages, in particular the Coqterms.

The λ-calculus of Coq is encoded by an Elpi type `term`, whose constructors used in this thesis are as follows:

```
type sort sort → term.
type fun name → term → (term → term) → term.
type prod name → term → (term → term) → term.
type app list term → term.
type global gref → term.
type pglobal gref → univ-instance → term.
```

Let us present these constructors in order.

First of all, sorts are encoded by a constructor `sort`. We distinguish the case `prop` that encodes the impredicative universe $\mathbb{P}$ from the case `typ I` that encodes the predicative universe $\square_i$ by associating i with `I` using another Elpi type representing universes.

Next, the encodings of the Π and λ binders are the `prod` and `fun` constructors respectively. This is also a HOAS encoding, where binders use meta-functions. As a result, there are no cases for bound variables or unification variables. Bound variables are represented with universal constants as in the previous example of simply-typed λ-calculus, and unification variables are represented by Elpi variables.[1] These two cases are notoriously difficult to deal with in the more traditional encodings of Coq terms, as these terms are tricky to handle. Indeed, these variables only make sense in a given context. To avoid having to take the names of bound variables into account in the reasoning, one prefers encodings that make α-equivalent terms syntactically equal. Thus, the common encodings of λ-calculus represent bound variables with De Bruijn *indices* [52], *i. e.*, a constructor $\texttt{var}_n$ where n is a natural number representing the distance to the binder whose variable this constructor points to. In this context, binders are just represented with the type of the bound variable and the bound term. For example, if we denote `lam` the constructor that encodes λ and `A` the encoding of a type A, the identity $\lambda x : A. x$ is encoded as `lam A` $\texttt{var}_1$, and the encoding of $\lambda x : A. \lambda y : A. x$ is `lam A (lam A` $\texttt{var}_2$`)`. The HOAS encoding chosen by Coq-Elpi is another way to represent terms up to α-equivalence.[2] Indeed, the use of a meta-function sets the position of the binder to which the variable refers, and even has the advantage over De Bruijn indices that it is impossible for a variable to escape its scope, whereas without further involvement of the type system of the meta-language, a De Bruijn index can be higher than the number of binders present in the term. Moreover, term abstraction or reduction requires rigour in index shifting, whereas these operations are trivial in Elpi.

1: A term with holes in Coq is therefore also a term with holes in Elpi.

[52]: De Bruijn (1972), "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem"

2: The first parameters of type `name` in the binders are just present for display reasons, but are not taken into account in Elpi's unification.

Concerning the remaining constructors, we note that application is n-ary to reflect the OCaml type of Coq terms, and constants are represented with two different constructors depending on whether they are universe-polymorphic. The names of constants are encoded in a `gref`[3] type that distinguishes between inductive type constructors — `indt` —, inductive value constructors — `indc` — and the rest of the definitions — `const`. Other term constructors are available, such as `match` or `fix`, but are not discussed in this thesis.

3: For *global reference*.

Here is an example of Coq term and its encoding in Elpi:[4]

```
fun (X : Type@{u}) (f : Type@{u} → A) ⇒ f X
```

```
fun `X` (sort (typ «u»)) x\
  fun `f` (prod `_` (sort (typ «u»)) _\ global (indt «A»)) f\
    app [f, x]
```

4: We assume here that `A` is a universe-monomorphic inductive type and `u` is a named universe.

A Coq-Elpi tool receives and sends Coq terms — command arguments, goals to prove, proof terms, *etc.* — in this format.

4.2 A toolbox

In addition to the features offered by the Elpi language and the high-level encoding of Coq terms, the Coq-Elpi plugin provides numerous entry points to the

Coq API, for example to search for the definition of a term in the Coq context in which the meta-program runs, call the typechecker, or create new definitions. Databases can also be created to store terms at the meta level, as well as commands and tactics to interact with the proof assistant, making Elpi a complete meta-programming tool.

In this section, we shall take the simple example of a term display feature, *i. e.*, the association of each type with a function making a string from a value of this type. To do this, the Haskell language uses a `Show` typeclass and includes an instance derivation feature for new data types with the `deriving` keyword:

```
class Show a where
  show :: a → String

data Maybe a = Just a | Nothing
  deriving Show
```

This feature can be obtained in Coq using a database storing the display functions specific to each constant and a command generating new functions from existing functions and a type declaration.

4.2.1 Databases

In Coq-Elpi, a database is a bank of predicate instances. A predicate can indeed be used as a way to group data or associate them together. In our case, we want to associate constants with Coq functions to display them. We can use a predicate `show` taking as arguments two constants, one for the type to be displayed, the other for the display function.

Database creation The creation of a Coq-Elpi database is done with the `Elpi Db` command, that is given the name of the database as well as a block of code declaring the various predicates representing the data that will be stored in it later. For example, here is the declaration of a database intended to contain instances of display functions:

```
Elpi Db show.db lp:{{
  pred show o:gref, o:constant.
}}.
```

Here, `show.db` is the name of the database. Note that the first argument can be any constant — inductive type, constructor, definition — while the second is necessarily a definition, since this is a function whose codomain is `string`.

Adding information to a database An instance of the `show` predicate is added to the database using a predicate internal to Coq-Elpi, `coq.elpi.accumulate`. It takes various arguments allowing the specification of the database in which to store the instance, at what location in this database more precisely, *etc.*, as well as the predicate instance. For example, if the user declares a term `show_nat` as a display function for type `nat`, it can be added to the database `show.db` with the following query in Coq-Elpi :

```
global Nat = {{ nat }},
global (const ShowNat) = {{ show_nat }},
coq.elpi.accumulate _ "show.db" (clause _ _ (show Nat ShowNat))
```

The {{ · }} syntax is the *quoting* operation enabling Coq syntax to be used within a Coq-Elpi program, the term then being translated into its Elpi representation. The syntax of the opposite operation, *unquoting*, allowing a term expressed in the meta-language to be written in the middle of a term expressed in Coq and to be evaluated to obtain a term entirely expressed in Coq, is `lp:{{ · }}`. Naturally, in a complete plugin for Coq, the user does not have to write this Elpi code manually to populate a database. Instead, they can use a Coq command provided by the plugin developer. This also has the advantage of adding an indirection level between user input and the actual database modification code, for example to check that the terms supplied by the user as arguments to the command are well typed and correspond to the format of the data we wish to store.

Database queries Once the user has filled the knowledge base with the desired information, any Coq-Elpi code connected to the base can exploit it by making queries. As the data are represented with predicate instances, performing a query amounts to calling one of the predicates defined in the database. If the instance we are looking for exists, then the predicate will succeed, otherwise it will fail.

4.2.2 Creation of commands and tactics

The main interest of a meta-language for Coq is to enable the development of tools to interact more efficiently with Coq, *i. e.*, commands and tactics. Commands allow registering data, to retrieve and display information, and to automate declarations. Tactics allow performing proofs more quickly than letting the user do it alone, by executing algorithms implemented in the meta-language to automate the proof steps. In the example given earlier, we want to have a command to register display functions and a command to generate new ones.

Commands A command allowing the user to declare display functions could be the following one:

```
Show Declare nat show_nat.
```

The main role of this command is to replace the manual call to the internal predicate `coq.elpi.accumulate`. It is declared as follows:

```
Elpi Command Show.
Elpi Accumulate Db show.db.
Elpi Accumulate lp:{{
  main [str "Show", str "Declare", trm T, trm F] :- % ...
}}.
```

After connecting the command to the database, we give the main predicate of the command, with the arguments it expects and the code to execute when it is called. Here, variable `T` will contain the term corresponding to the type for which we want to add a display function. This function is then represented by variable `F`. This main predicate calls `coq.elpi.accumulate`, but it can check beforehand that `T` and `F` are well typed, for example, or even that `F` has the type of a display function for `T`. This ensures that the database contains well-formed data at all times.

A command can be declared in a similar way to generate a display function on an inductive type `I` from its definition. An internal predicate `coq.env.indt` is then called to retrieve the declaration, giving amongst other things the types of

the inductive type constructor and the inductive value constructors. The function can then be generated by syntactically inspecting these types. Once the display function has been built, the internal predicate `coq.env.add-const` can be called to declare this term as a new constant `show_I` for example.

Tactics A tactic is created with Coq-Elpi using the following command, where `t` is the name of the tactic:

```
Elpi Tactic t.
```

Just like commands, a tactic has a main predicate,[5] defined by accumulating Elpi code, after connecting the tactic to the various databases its code uses:

5: In the case of tactics, it is called `solve`.

```
Elpi Accumulate lp:{{
  solve InitialGoal NewGoals :- % ...
}}.
```

The `solve` predicate takes the initial goal and must return a list of goals containing the associated goal as well as any proof obligations that the tactic leaves behind.

The initial goal is represented with a value of the Elpi type `goal`, containing the context of the goal, the type to inhabit in order to prove it, and the various arguments given to the tactic. This `goal` type also contains variables of type `term` representing the proofs to apply for proving the goal. These are initially undefined variables, and any unification constraint applied to them results in an action on the proof term in Coq. Touching these terms directly is therefore a tricky business, but an API is available in Coq-Elpi to act on the proof term in a controlled manner. In the context of this thesis, we only use the `refine` function that essentially performs the same action as the Coq tactic with the same name, *i. e.*, applying a proof term with holes, these holes representing the new proof obligations. Coq checks that the term is well typed, provided that the user subsequently fills the holes.

Trakt:
Proof transfer
by canonisation

Introduction

The decision procedures used to automate the proofs of statements contained in a theory are defined in an abstract way, using the mathematical objects present in the *signature* of this theory, *i. e.*, the list of symbols belonging to the theory as well as the various equations governing it. However, in the Coq proof assistant, it is possible to model the same mathematical object in different ways, as shown by the definition of inductive types `nat` and `bin_nat` in § 2.2.2. This diversity in the representation of mathematical objects is available both for the tactics developer and for the user of the proof assistant.[6] For example, when implementing a decision procedure for Coq, the developer chooses a representation for each of the symbols of the concerned theory. The automation tactic — the result of this implementation — is then initially biased towards one of the representations: if the user chooses the same data structures as the developer of the tactic to represent the theory, then the decision procedure can run normally and provide the expected automation; otherwise, the tactic does not recognise the different symbols, which makes it unable to fulfil its role correctly.

6: For the latter, it is important, as certain proofs are more easily done with certain representations.

In order to increase compatibility of the goals with the decision procedures, one solution is to add a preprocessing phase between these goals and the associated tactics. An ideal preprocessed statement would express all the symbols of a theory's signature with terms defined using the same data structure, the one recognised by the automation tactic that we wish to execute after preprocessing. The various representations that can be found in the statements must therefore all *converge* to the target signature. The preprocessed statements are then expressed in a canonical form maximising the chances of success of the automation tactic that is subsequently executed. In order to replace the original goal with the preprocessed goal, it is necessary to prove that the latter implies the former. Indeed, if we wish to prove G' when Coq expects a proof of G, we must apply the *modus ponens* rule using a function of type $G' \rightarrow G$, that performs the goal substitution.

This section presents Trakt[7] [12], a pragmatic goal preprocessing plugin for Coq whose goal is to allow more statements to be proved by existing implementations of decision procedures. We first present the specification of the desired preprocessing tool and we position the existing tools in relation to this objective (§ 5). We then give a detailed theoretical presentation of the preprocessing carried out by Trakt (§ 6). Finally, we position this tool in relation to an ecosystem of preprocessing tools in Coq, and identify the prospects for improvement (§ 7).

7: The implementation is available in the repository:

https://github.com/ecranceMERCE/trakt

[12]: Blot *et al.* (2023), "Compositional preprocessing for automated reasoning in dependent type theory"

Goal canonisation: objectives and current situation 5

The context of goal preprocessing in Coq is broad, and the nature of the rewrites to perform depends on the decision procedure we wish to execute after preprocessing, as well as the flexibility of its implementation. We restrict this context to goal *canonisation*, *i. e.*, exploitation of equivalences between types and operations in a rewriting of the goal, so that it uses the same implementation of the signature of a theory as the one chosen by the developer of the automation tool we wish to run to prove the goal. However, even within this reduced frame, perfect preprocessing does not exist, as each tool deals with a specific sub-problem and not necessarily in a complete way. The development of preprocessing tools is therefore incremental, each tool being justified by the identification of limitations in existing tools dealing with the same problem. In this chapter, we define the preprocessing problem we are interested in for the first part of this thesis (§ 5.1), then we study the features of a family of preprocessing tools based on the `zify` tactic, as well as their limitations regarding the problem to solve (§ 5.2).

5.1 Content of the desired preprocessing algorithm 43
5.1.1 Preprocessing of theories . . . 43
5.1.2 Status of logic 45
5.1.3 Polymorphism and dependent types 46
5.2 The `zify` family: features and limits 46
5.2.1 Modular preprocessing of arithmetic 47
5.2.2 Preprocessing of logic 48
5.2.3 The `mczify` extension 49
5.2.4 Limitations of `zify` 49

5.1 Content of the desired preprocessing algorithm

An efficient way to define the objective of a goal preprocessing tool is to base it on the automation tactic that we wish to execute to prove the statements contained in a given theory, and to identify the differences between the goals frequently encountered by the user in this theory and the implementation of the signature of the theory in the automation tactic. In the case of Trakt, the general objective is to have a preprocessing tool for theories in the SMT family, and in particular to improve the preprocessing phase of the SMTCoq plugin [41], a tool for connecting Coq to SMT solvers. However, we want the tool to be compatible with other automation tactics, such as `lia`, by making it flexible on the preprocessed theory and not hard-coding a particular signature in the plugin. In this section, we explain all the objectives of the canonisation expected in this context.

[41]: Ekici *et al.* (2017), "SMTCoq: A Plug-In for Integrating SMT Solvers into Coq"

5.1.1 Preprocessing of theories

The first objective of canonisation is to erase variability in the representations of mathematical objects in Coq, due to the freedom and expressiveness of the proof assistant language. This amounts to translating a goal that uses one or more implementations of the signature of a theory into another goal that uses one implementation of the signature defined as canonical. This implies knowing how to deal individually with all the possible elements of a signature and their occurrences in the goal. The examples consider the automation tactic `lia` [17] for linear arithmetic over integers.

[17]: Besson (2006), "Fast Reflexive Arithmetic Tactics the Linear Case and Beyond"

Data types The canonical implementation selected by `lia` for the signature of linear integer arithmetic is based on the `Z` type of binary integers:

```
Inductive Z : Set :=
  | Z0 : Z
  | Zpos : positive → Z
  | Zneg : positive → Z.
```

The chosen representation uses type `positive` defined in § 2.2.2.[1] As this is the canonical type for integers, a value in `Z` is recognised as an integer in any execution of the `lia` tactic. Conversely, integers expressed in other encodings are not recognised as such, and may cause the tactic to fail.[2] For example, the MathComp library provides a type `int` with a unary encoding that glues two copies of the space of natural integers together:

1: It is in fact `bin_nat` with an additional case for non-zero negative integers.

2: Here, we consider a version of `lia` without preprocessing. In the standard version of Coq, this tactic comes with integrated preprocessing, which Trakt improves.

```
Variant int : Set := Posz of nat | Negz of nat.
```

Although this multiplicity of representations offers greater freedom to the user, the tactics remain rigid due to the choices required for implementation. The `lia` tactic does not — by default — treat the values in `int` as integers. The desired preprocessing phase replaces values from various encodings with values in a target encoding. In the case of arithmetic, for example, we want to replace values in `int` with values in `Z` to improve `lia`'s support of the goal. In the case of quantified statements, we also want the type of bound variables to change accordingly, at least for quantifications over simple types.

Operations and constants In the desired preprocessing, the replacement of values in a source type by values in a target type must be carried out at a finer level of granularity. It must be possible to associate an operation in the source type with an equivalent operation in the target type. Otherwise, a value `a + b` in a source type — for example `int` — could be translated as a single block, which would no longer make it an addition in the target type. If the operations are associated together, then the preprocessing tool can translate first the operation, then its arguments. This treatment also applies to constants in the source type, which can be considered as zero-arity operations. If the constructors of a source type can be associated with constants and operations in the target type, then numerical constants can be translated.

Subtyping Some goals encountered in proofs may contain types that can be embedded in a larger target type, without being equivalent to them. This is the case, for example, of type `nat` and type `Z`. Indeed, it can be interesting to visualise natural numbers as a special case of integers in order to facilitate the work of a proof automation tool like `lia`. We then expect the translated goal to contain a property that makes the subtyping information explicit, so that this associated statement remains provable if the original statement is.

To sum up, we consider the following goal:

```
forall (a b : nat), a + b = b + a
```

We need to be able to translate this goal into the following goal:[3]

3: We use the same notation `+` for addition in both types, for readability.

```
forall (a b : bin_nat), a + b = b + a
```

We can see that the values are treated independently: additions are conserved in the preprocessed goal and quantifiers are updated with bound variables in the target type. If the automation tool requires the values to be expressed in a larger target type, we expect a goal similar to this one, adding conditions to preserve information about the positivity of `a` and `b`:

```
forall (a b : Z), a ⩾ 0 → b ⩾ 0 → a + b = b + a
```

5.1.2 Status of logic

The preprocessing tool we want to build focuses on translating the symbols of a theory, but when translating a goal, we cannot ignore the processing of logic, expressed through quantifications, relations, logical connectives, *etc.* Even in a simple goal, the user needs logic to express what they want to prove.

Relations For example, in the last goal presented earlier, the property to prove is an equality between two arithmetic expressions. The inductive type `eq` used to represent equality is a binary relation on a given type. In the associated goal, equality was changed into equality over the target type. We are therefore looking to build a preprocessor that also substitutes relations. Note that this must be extended to relations of arbitrary arity, including predicates, of arity 1.

Representations of logic The expressiveness of Coq allows logic to be expressed in a variety of ways, not all of which are necessarily handled by proof automation tactics. In general, properties can be expressed using inductive types in `Prop` or `Type`. However, a user or library developer may use relations in other representations in their goals, that are not recognised by all automation tactics. This is because decidable relations are generally encoded as booleans, on which it is easy to perform a case analysis.[4] The types in the MathComp library [35] come with definitions allowing them to be used with boolean logic — for example, an equality test or an order relation. These boolean encodings are less likely to be recognised by an automation tactic than their versions in `Prop` defined in the Coq standard library. Preprocessing able to express logic in `Prop` is therefore of interest.

4: As the logical system of Coq is not classical, it is not always possible, by default, to perform a case analysis on a value in `Prop`.

[35]: Mahboubi *et al.* (2021), *Mathematical Components*

Conversely, if an automation tactic involving a lower-level logic system than that of Coq is used, in which every relation is decidable, then the goal it receives must expose the decidability information of the relations as much as possible. For example, the SMTCoq project, targeting SMT solvers, directly maps the boolean logic of Coq to the logic of the SMT-LIB language used as the input format for the solvers. Thus, the optimal input goal for the SMTCoq automation tactic is one where all logic is expressed in `bool`.

Logical connectives Coq users can declare new logical connectives, for example an exclusive OR, and use them in their goals. If these connectives have a boolean version, the preprocessing tool must be able to target one version or the other, so that the entire logical part of the syntax tree representing the goal is expressed in the desired representation of logic.

To summarise, let us consider the following goal, expressed using boolean logic and MathComp integers:[5]

5: A coercion from `bool` to `Prop` defined in this library for greater readability must also be taken into account by the preprocessing tool.

```
forall (x y z : int), x ⩽? y && y ⩽? z ⟶ x ⩽? z
```

Order relations, implication, and conjunction are expressed in a boolean version. The desired preprocessing is able to translate this statement by targeting both another representation of integers, such as `Z`, and an expression of logic in `Prop`.

```
forall (x y z : Z), x ⩽ y ∧ y ⩽ z → x ⩽ z
```

5.1.3 Polymorphism and dependent types

The expressiveness of Coq allows the user to prove very general statements that are true for a family of types. For example, we can prove statements about a data structure regardless of its size or the type of elements it contains. The preprocessing features defined previously must then be extended to this larger context, so that these more abstract statements can also be proven by automation tools.

Polymorphism If we know how to associate a value in a type `A` with a value in a type `B` in the output goal, it may be interesting to lift this association to a data structure containing values in `A`, in order to obtain in the associated statement a similar structure with its values in `B`. Thus, a list of integers `list int` can be associated with a list of integers `list Z` in the associated goal, which then allows a tactic capable of handling list theory and arithmetic in `Z` to prove the goal.

Without considering the exact type of values contained in a data structure, we can also perform logical preprocessing, for example by propagating a decidable equality through the structure to obtain one on the structure itself. The associations between relations detailed earlier are then parameterised by other relations.

It can also be useful to associate several data structures together. For example, turning a list of integers in type `list int` into a tree of integers in type `tree Z` can be done by reorganising the values in the list.

Finally, some forms of *ad hoc* polymorphism may be present in goals, such as the generic notations of the MathComp library, which allow the same notation to be used for several types equipped with the same operation. For example, the addition of MathComp is noted `+` for all the types that actually are instances of a ring mathematical structure, encoded by canonical structures.[6] This genericity must not hinder the operation of the preprocessing tool.

6: Concept presented in § 3.1.2.

Dependent types Besides polymorphism, some data structures are indexed by values that are not types. For example, we can imagine a type `bitvector n` that represents bit vectors of size `n`, where `n` is a natural integer in type `nat`. In this case, the preprocessing of a goal containing this type is more subtle. If the data structure does not change during preprocessing, then value `n` must remain in the source type `nat`, as a change in the type would make the associated goal ill typed. If the structure changes, for example for another structure indexed by an integer in `Z`, then the user must be allowed to associate such dependent structures.

5.2 The `zify` family: features and limits

The `zify` [53] tactic is a tool for preprocessing arithmetic and logical statements for Coq, designed for the `lia` tactic. It is the starting point for the work on Trakt and a source of inspiration for certain design choices. It translates some integer representations (`nat`, `positive`, *etc.*) to type `Z`, the representation of integers used by the target tactic. This tactic is certifying and does not leave proof obligations to the user. In this section, we present the internals of this tool and evaluate its response to the specification defined in the previous section.

[53]: Besson (2017), "ppsimpl: a reflexive Coq tactic for canonising goals"

5.2.1 Modular preprocessing of arithmetic

Modularity with respect to a single, fixed signature for a theory can be represented by a Coq record type or a module type. Thus, for arithmetic, we could have a record defined by a carrier type, the operations of addition, multiplication, *etc.* However, to allow the user to only partially preprocess the theory, or to add new symbols, *i. e.*, new arithmetic operations to be taken into account, the `zify` tactic offers a degree of extensibility in the form of typeclasses.[7] In this way, there is a typeclass for each category of symbols to preprocess.

7: Concept presented in § 3.1.2.

All type classes instances are then exploited when traversing the original goal, each symbol being replaced with its associated symbol, adding a subterm to the global proof, which is eventually a proof of implication between the new goal obtained and the initial goal.

Type embeddings The first class, `InjTyp`, is used to declare an *embedding* between two Coq types:

```
Class InjTyp (S T : Type) := {
  φ : S → T;
  P : T → Prop;
  π : forall (x : S), P (φ x)
}.
```

Function `φ` is an injection from the source type `S` to the target type `T`. The injection can be partial thanks to the following two fields: `P` is a property on the values of the target type, and `π` is a proof that any injected value respects this property. This allows representing subtyping, such as the one of `nat` in `Z` mentioned earlier. In the case of two equivalent types, such as `int` and `Z`, the fields `P` and `π` are filled in a trivial way, for example the following:

```
P _ := True
π _ := I
```

In the case of the *partial* embedding of `nat` into `Z`, the property is positivity:

```
φ := Z.of_nat
P (n : nat) := Z.of_nat n ⩾ 0
π : forall (n : nat), Z.of_nat n ⩾ 0
```

Symbol embedding In order for preprocessing to replace an operation with the associated canonical operation, classes are used for various arities: `CstOp`, `UnOp`, `BinOp`. Here is the definition of the `BinOp` class:

```
Class BinOp
  {S₁ S₂ S₃ T₁ T₂ T₃ : Type}
  `{InjTyp S₁ T₁} `{InjTyp S₂ T₂} `{InjTyp S₃ T₃}
  (op : S₁ → S₂ → S₃)
    := {
      op' : T₁ → T₂ → T₃;
      π₂ : forall (x₁ : S₁) (x₂ : S₂), φ (op x₁ x₂) = op' (φ x₁) (φ x₂)
    }.
```

One links a binary operation op, defined from three types declared as embeddable, to an operation op', defined with the three associated target types, by a proof of morphism π_2.

Example 5.2.1
Here is the declaration of an embedding of addition over nat into addition over Z:

```
op' := Z.add
π2 : forall (n1 n2 : nat),
        Z.of_nat (n1 + n2) = Z.of_nat n1 + Z.of_nat n2
```

5.2.2 Preprocessing of logic

The way to handle logic in zify is similar to the arithmetic preprocessing. Just as there are typeclasses for operations, the plugin defines typeclasses for various arities of relationships.

Relation embedding The zify tactic is also able to preprocess homogeneous binary relations thanks to the BinRel class:

```
Class BinRel {S T : Type} (R : S → S → Prop) `{InjTyp S T} := {
  R' : T → T → Prop;
  πR2 : forall (x1 x2 : S), R x1 x2 ↔ R' (φ x1) (φ x2)
}
```

So, for instance, an order relation over type positive can be embedded into the order relation over Z:

```
R' := Z.ge
πR2 : forall (p1 p2 : positive), p1 ≥ p2 ↔ Zpos p1 ≥ Zpos p2
```

General processing of logic Other features are proposed, such as a saturation mechanism, so as to recover properties lost during embedding, such as preservation of positivity by multiplication over natural integers, or the possibility to declare morphisms for equivalence in Prop, to allow various logical connectives to be handled. The preprocessing of logic in general is possible *via* the use of the BinRel and BinOp classes. Indeed, boolean connectives must be declared as binary operators of the BinOp class (with S_3 := bool) and associated either to themselves, to their version in Prop thanks to a trivial embedding of bool into Prop, or to operators in an integer type, as some goals use boolean logic in the middle of arithmetic computations.

Example 5.2.2
With zify, it is possible to preprocess the following goal:

```
forall (n : nat), n + n ≥ n
```

The obtained associated goal is then the following one:

```
n : nat      pn : Z.of_nat n ≥ 0
---------------------------------
Z.of_nat n + Z.of_nat n ≥ Z.of_nat n
```

Here, the p_n property represents the partiality of the embedding of n into Z.

The comparison operator used in the output goal is the one of type `Z`.

5.2.3 The `mczify` extension

There is an extension to `zify` called `mczify` [54], whose purpose is to manage the various integer types defined in the MathComp library, as well as the associated operators. It was built thanks to the extensible nature of `zify`, since it consists of a series of declarations of instances of the type classes presented previously. The declarations linked to the MathComp operators make `mczify` a masterpiece, as they are generic operators.

[54]: Sakaguchi (2019–2022), *Micromega tactics for Mathematical Components*

Example 5.2.3
Thanks to the `mczify` upper layer, the following goal can be handled by `zify`:

```
forall (x : int), x ⩾ 1 → x * x ⩾ x
```

The `x * x` subterm is actually `@GRing.mul int_Ring x x`, where `GRing.mul` is the projection used to get the multiplication operation from the ring instance `int_Ring` declared in the MathComp library for type `int`. In the same way, `x ⩾ 1` is actually `@Order.ge int_porderType x (@GRing.one int_Ring)`, where `int_porderType` is the instance of the MathComp structure representing order relations for type `int`. Note that the value `1` is also a field of the ring structure. The declarations added by `mczify` thus allow using MathComp's *ad hoc* polymorphism on the theory of arithmetic and on logic, while remaining compatible with `zify`.

5.2.4 Limitations of `zify`

It is clear that the `zify` tactic fulfils its role as a preprocessing tool for `lia`, as the base declarations present in the plugin for the various type classes presented target the exact set of goals recognisable by the automation tactic. The tool, like the target tactic, is able to step under the various quantifiers and logical connectives present in the goal, as well as preprocess the operations and signature values of Presburger arithmetic, this for the integer types of the standard library, *i. e.*, the most frequently used. Thanks to its extensibility, the plugin is not limited to these reference types. As the `mczify` extension shows, it can be extended to the types and operators of a custom library, including those of a certain complexity.

However, this *ad hoc* tool can hardly be used in a different context. For instance, within the class of decidable goals, which are frequent in proofs and not very interesting for humans to prove,[8] lie the SMT problems, in which *uninterpreted symbols* appear, and in particular, functions.[9] Several plugins for Coq can be used to send these statements to SMT solvers, such as Itauto [55] or SMTCoq, but these interfaces also have an input signature that must be respected.[10] In this sense, they suffer from the same problem as the `lia` tactic. However, for such statements, the tools of the `zify` family are insufficient by design, as they were not created with the aim of handling these cases going beyond the logical fragment provable by `lia`. For example, in the case of an uninterpreted function `f : int → int` applied to an integer `x`, the tactic `zify` treats term `f x` as a single value of type `int`. This situation is acceptable in the context of the use of `lia`, as the behaviour of the tactic is aligned with that of `zify`, but if the associated statement is given to an SMT solver, it may fail to prove it because of this information loss during preprocessing.

8: It is the class of goals where automation is the most expected.

9: This theory is called theory of equality, theory of congruence, or in SMT vocabulary, UF theory (Uninterpreted Functions).

[55]: Besson (2021), "Itauto: An Extensible Intuitionistic SAT Solver"

10: The SMTCoq project has a preprocessing phase but it is basic and not very extensible.

For example, in the case of an automation tool that would need to remove any logical element expressed in `Prop`, the `zify` tactic would perform incomplete processing. Indeed, although relations on integers can be replaced with their boolean counterpart, a logical connector in `Prop` cannot be turned boolean, because the `BinRel` class would in this case require `Prop` to be embeddable into `bool`, which is not true in the general case. It seems interesting to give the preprocessing tool features for logic that are independent from what is done on the signatures of theories.

Another conceptual limitation of `zify` is the use of Coq terms to store user information. Using typeclasses allows exploiting the inference features of Coq, but forces the developer to model their data in a Coq type, which can lack flexibility. For example, a typeclass is needed for each possible arity on operations and relations. In addition, it can become difficult to capture the various possible associations between types and symbols in a same Coq type, especially if we want to build a preprocessing tool that supports polymorphism and dependent types, which is not the case of `zify`. As a result, there is a gap to be filled between the initial goals entered into Coq by the user and proof automation tools.

Example 5.2.4
The following goal belongs to the UFLIA[11] theory:

```
forall (f : int -> int) (x : int), f (2 * x) <=? f (x + x)
```

It is theoretically provable by an SMT solver, but it is outside the fragment accepted by the tactics of the `zify` family. Without preprocessing, the plugins in charge of delegating proofs to an SMT solver cannot prove it. The aim of Trakt is to bridge the gap between this kind of goals and these plugins.

11: Uninterpreted Functions and Linear Integer Arithmetic.

Theoretical mode of operation 6

6.1 Gathering user information . 51
6.1.1 Type embeddings 51
6.1.2 Logical embeddings 52
6.1.3 Symbol embeddings 53
6.1.4 Conversion keys 53
6.2 Preprocessing algorithm . . . 54
6.2.1 Handling universal quantifiers 54
6.2.2 Handling logical connectives . 56
6.2.3 Theory-specific preprocessing 58
6.2.4 The `trakt` tactic 59

The first prototype developed in this thesis, Trakt [12], is a tool for preprocessing by goal canonisation in Coq, with the aim of extending the preprocessing carried out by the `zify` tactic to statements from the SMT family. We therefore want to go beyond the limits of `zify` without introducing any regression in the features already present. This chapter looks at how canonisation is carried out in Trakt. First, we list the kinds of information that the user can provide to the tool (§ 6.1), then we detail the algorithm used to solve the theoretical problem (§ 6.2).

[12]: Blot *et al.* (2023), “Compositional preprocessing for automated reasoning in dependent type theory”

6.1 Gathering user information

Canonisation is only made possible by the knowledge of the various existing embeddings that start from the terms found in the input goal given to the preprocessing tool. Trakt allows different kinds of information to be declared: type embeddings, logical embeddings, symbol embeddings, and conversion keys.

6.1.1 Type embeddings

As explained in § 3.3.2, to perform proof transfer, it is necessary to define a relation between source and target types. In Trakt, this relation is bijection, and it is defined for inductive types with no parameters or indices, such as integer types. We declare that a type A can be embedded into a type B when there is a bijection between them:[1]

$$\Sigma(\phi : A \to B)(\psi : B \to A).\,(\psi \circ \phi \doteqdot \mathrm{id}) \times (\phi \circ \psi \doteqdot \mathrm{id})$$

A command is provided for the user to declare this information:[2]

```
Trakt Add Embedding A B φ ψ π₁ π₂.
```

This declaration allows embedding any value of type A or any functional type containing A into the associated target type. For example, a function of type $A \to A$ can be embedded as a value of type $B \to B$. In particular, Trakt is able to handle uninterpreted symbols present in statements of the SMT class, as well as universally quantified variables having a type eligible for embedding.

1: We denote as $\doteqdot$ pointwise equality, *i. e.*,

$$f \doteqdot g \quad := \quad \Pi x.\, f\, x = g\, x$$

2: Here, π_1 and π_2 represent the section and retraction proofs showing that φ and ψ are inverses of each other — respectively the first and second proofs of the pair under the Σ-type here on the left.

Partial embeddings There are relevant instances of preprocessing for which the two embeddings functions are not inverses. This is the case for the embedding of $\mathbb{N}$ into $\mathbb{Z}$ for example, where the pseudo-inverse embedding function is not injective since it associates a default value to negative integers, as they have no equivalent in the space of natural numbers. In this case, we can only prove a weakened version of retraction, in which the property is only true on values for which ψ does not truncate, *i. e.*, if there is an antecedent in A for these values. In the case of the embedding of $\mathbb{N}$ into $\mathbb{Z}$, this condition called *embedding condition* will be positivity. The embedding condition is represented by additional data

in the proof supplied by the user: the condition and a proof that any embedding from A respects it. A partial embedding is therefore defined as follows:

$$\Sigma(\phi : A \to B)(\psi : B \to A)(P : B \to \mathbb{P}).$$
$$(\psi \circ \phi \doteqdot \mathrm{id}) \times (\Pi b : B.\, P\, b \to \phi\,(\psi\, b) = b) \times (\Pi a : A.\, P\,(\phi\, a))$$

The command to add type embeddings to Trakt accepts partial embeddings:[3]

```
Trakt Add Embedding A B φ ψ P π₁ π₂P πP.
```

3: Here, π_{2P} is the proof of conditional retraction and π_P is the proof that any embedding from A verifies the embedding condition.

When embedding a variable, the condition will be made explicit so that it is always possible to prove that the final goal implies the initial goal. Thus, an uninterpreted function $f : \mathbb{N} \to \mathbb{N}$ will be replaced with a function $f' : \mathbb{Z} \to \mathbb{Z}$ packed with the following property:

$$\Pi x : \mathbb{Z}. \quad x \geq 0 \to f'\, x \geq 0$$

6.1.2 Logical embeddings

In a statement of the SMT class, the subterms contained in one of the processed theories are atoms in the tree that represents the logical formula, the nodes of this tree being the logical connectives, equalities, and other predicates of arbitrary arity. Depending on the automation tactic targeted after the use of Trakt, these predicates must also be translated, either to a boolean version or to a version in $\mathbb{P}$. Trakt allows declaring an embedding between two predicates P and Q of the following types, where each type T_i' is either T_i itself, or an embedding from T_i:

$$P : T_1 \to \cdots \to T_n \to L$$
$$Q : T_1' \to \cdots \to T_n' \to L'$$

The declaration is made by proving an equivalence between both predicates:

$$\Pi x_1 \cdots x_n. \quad P\, x_1 \cdots x_n \;\; \bowtie_{L,L'} \;\; Q\,(\phi_1^?\, x_1) \cdots (\phi_n^?\, x_n)$$

L	L'	$\bowtie_{L,L'}$
$\mathbb{B}$	$\mathbb{B}$	$=$
$\mathbb{P}$	$\mathbb{B}$	$\lambda P\, b.\;\; P \leftrightarrow b = 1_{\mathbb{B}}$
$\mathbb{B}$	$\mathbb{P}$	$\lambda b\, P.\;\; b = 1_{\mathbb{B}} \leftrightarrow P$
$\mathbb{P}$	$\mathbb{P}$	$\leftrightarrow$

The logical codomains of the predicates L and L' are either $\mathbb{P}$ or $\mathbb{B}$, and $\bowtie_{L,L'}$ is a way to express equivalence depending on these codomains. $\phi_i^?$ designates an embedding function between T_i and T_i' that is optional at the meta level: if both types are identical, this term is absent.

Example 6.1.1
In the case of an embedding of equality over type `int` into boolean equality over `Z`, the proof to be declared to Trakt has the following type:

```
forall (x y : int), x = y <-> Z_of_int x =? Z_of_int y = true
```

In this goal, `Z_of_int` is the embedding function from `int` to `Z` and `=?` is the boolean equality over `Z`.

The command available to the user for declaring logical embeddings is:

```
Trakt Add Relation n P Q π_L.
```

Value n is the arity of the declared predicates,[4] P and Q are the source and target relations, and π_L is the proof of equivalence between the two predicates.

4: This information is required so that it is possible to declare particular relations expressed in several terms, such as equality, which takes a type argument.

6.1.3 Symbol embeddings

The basis of a mathematical theory is often a set with operations. The signature of a theory in Coq is therefore a type called *carrier* along with values and operations defined over this type. Type embedding makes it possible to manage variability on the carrier types in the various goals, but the symbols from the signature still need to be processed. To that end, Trakt also makes it possible to declare embeddings between the various symbols, of any arity, by giving a source symbol and a target symbol, as well as the morphism property:

$$s : T_1 \to \cdots \to T_{n+1}$$
$$s' : T_1' \to \cdots \to T_{n+1}'$$

$$\Pi x_1 \cdots x_n. \quad \phi^?_{n+1}\,(s\,x_1\,\cdots\,x_n) = s'\,(\phi^?_1\,x_1)\,\cdots\,(\phi^?_n\,x_n)$$

Example 6.1.2
We can embed addition over type nat into addition over Z by giving a proof of the statement from Example 5.2.1, and the zero of nat into the one of Z by making explicit for Trakt that Z.of_nat O = Z0.

The declaration of a symbol is done through the following command:

```
Trakt Add Symbol S S' π_S.
```

Values S and S' are the source and target symbols, and π_S is the morphism proof.

6.1.4 Conversion keys

For performance reasons in the implementation, the default term recognition in Trakt is purely syntactic. However, in order to keep the additional features provided by mczify and presented in Example 5.2.3, if the user declares embeddings of concrete operations that are then packaged into structures, the preprocessing tool must be able to detect that generic projections available for these structures yield the same terms as the ones previously declared. It is therefore necessary to be able to use Coq conversion locally for these terms, that we call *conversion keys*.

In Example 5.2.3, if the user declares an embedding from concrete multiplication over type int, then using generic notations in the goals should not impact preprocessing. Therefore, @GRing.mul int_Ring must be preprocessed as if the concrete operation had been used there, which is possible by declaring GRing.mul as a conversion key. Such behaviour is possible because the projection reduces precisely to the concrete operation. The type of the proof will thus not be invalid.

The command used to declare conversion keys is the following:

```
Trakt Add Conversion K.
```

where K is the term to declare as a conversion key.

6.2 Preprocessing algorithm

Using all the information provided to Trakt by the user, the tool is able to preprocess the input goal. This section details how the preprocessing algorithm works on universal quantifiers, logical connectives and the subterms belonging to a theory, before presenting the `trakt` tactic that implements it.

The algorithm takes a goal to be preprocessed as input, and is parameterised by the target type desired by the user to express logic as well as the target type for the theory to be preprocessed in the goal. It then generates an output goal as well as a proof that it implies the input goal. Due to the polarity of logical connectives, on some subterms this proof of implication must be generated in the opposite direction, a subtlety handled by the algorithm.

Unless explicitly stated, in the examples, we consider an embedding towards type `Z` and logic expressed in `Prop`.

6.2.1 Handling universal quantifiers

The first construction encountered in a goal to be preprocessed is often a universal quantifier. This case is handled by a recursive call on the subterm, yielding a proof, and a combinator to extend this proof to the quantifier. On the other hand, the type of the variable bound in the new goal must make maximum use of the type embeddings declared by the user, functional types also being taken into account, as explained in § 6.1.1. The proof generation process therefore acts differently depending on the type of the bound variable before and after translation, and the polarity at the time of translating the quantifier.

Unchanged type If the type of the bound variable does not change, then we must prove

$$\Pi x : A.\, B' \quad\to\quad \Pi x : A.\, B$$

from the proof $p : B' \to B$ obtained on the subterm.[5] The combinator to use is the following, regardless of the polarity:

$$\lambda(H : \Pi(x : A).\, B')(x : A).\, p\,(H\,x)$$

5: The contravariant case swaps B and B' in the types of the proofs.

Embedded type, covariant case If the type changes, then all occurrences of variable x are subject to a $\phi_{A\rightsquigarrow A'}$ embedding when the subterm is translated.[6] The output subterm at the level of the quantifier is therefore obtained by ignoring these embedding functions, *i. e.*, by replacing all occurrences of $\phi_{A\rightsquigarrow A'}\ x$ with a variable x' of the same type A', in order to obtain a new subterm depending only on the new variable x' and to be able to close the quantified term. Thus, in the covariant case, the proof to be provided at the level of the quantifier is

6: Here, $\phi_{T\rightsquigarrow T'}$ is a combinator adding all the necessary embedding functions from the potentially functional type T to the associated type T'.

$$\Pi x' : A'.\, B'' \quad\to\quad \Pi x : A.\, B$$

and we start from a proof $p : B' \to B$.[7] Here, B'' is the translation of subterm B' in which we have replaced the embeddings of x with x'.[8] Consequently, the combinator can instantiate the hypothesis with the embedding of variable

7: The case of partial embeddings is explained in the next paragraph.

8: $B'' \equiv B'[\phi_{A\rightsquigarrow A'}\ x := x']$

x, providing a proof of $B''[x' := \phi_{A \rightsquigarrow A'}\, x] \equiv B'$. Proof p can then be applied directly. The new combinator is as follows:

$$\lambda(H : \Pi(x' : A').\, B'')(x : A).\, p\, (H\, (\phi_{A \rightsquigarrow A'}\, x))$$

In the case where explicitly casting x from A to A' involves a partial embedding, the bound variable x' in the output goal must come with a property $P\, x'$ combining various embedding conditions defined in § 6.1.1. The proof to be provided is therefore the following:

$$\Pi x' : A'.\, P\, x' \to B'' \quad \to \quad \Pi x : A.\, B$$

As the combinator instantiates x' with the embedding of x, the property is always true, by composition of the proofs supplied by the user.[9] If we denote this proof combination π^*_P, the combinator in the case of partial embeddings is the following:

9: Value π_P in each declaration of partial embedding.

$$\lambda(H : \Pi(x' : A').\, P\, x' \to B'')(x : A).\, p\, (H\, (\phi_{A \rightsquigarrow A'}\, x)\, (\pi^*_P\, x))$$

Consider the following statement, representing a proof that a function is constant by recurrence on its domain, natural numbers:

```
forall (f : nat → nat) (k : nat),
  f 0 = k → (forall (n : nat), f (S n) = f n) →
    forall (n : nat), f n = k
```

During a preprocessing phase where `nat` has been declared embeddable into `Z`, the first quantifier is changed into a new bound variable `f' : Z → Z` along with a property combining twice the embedding condition of `nat` into `Z`:

```
forall (x' : Z), x' ⩾ 0 → f' x' ⩾ 0
```

In the proof for the quantifier, this property is proved thanks to the fact that the concrete value for `f'` is a composition of `f` with the embedding functions between `nat` and `Z`:

```
f' := fun (x : Z) ⇒ Z.of_nat (f (Z.to_nat x))
```

In particular, any application of `f'` to a term `t` can be seen as the application of `Z.of_nat` to `f (Z.to_nat t)`, so it is positive thanks to the proof of the embedding condition given when declaring the embedding from `nat` to `Z`:

```
forall (n : nat), Z.of_nat n ⩾ 0
```

Embedded type, contravariant case[10].

10: Here, we only deal with the case of partial embedding, the other cases can be obtained by simplifying it.

The contravariant case removes the need to prove the embedding conditions, as the implication has to be proved in the other direction, and they become additional hypotheses rather than arguments to be provided to use a hypothesis. However, these conditions are still useful for eliminating embedding identities appearing when hypotheses are instantiated. Indeed, the proof to build has the following type:

$$\Pi x : A.\, B \quad \to \quad \Pi x' : A'.\, P\, x' \to B''$$

The combinator therefore has at its disposal a variable $x' : A'$ as well as a proof of $P\, x'$, and it must instantiate a hypothesis $H : \Pi(x : A).\, B$. The solution is the inverse embedding of x', that can be written as $\psi_{A \leftsquigarrow A'}$. We then obtain a

proof of $B[x := \psi_{A \leftsquigarrow A'}\ x']$, knowing that we have the proof on the subterm $p : B \to B'$. Note that p is built from variable x obtained by going under the quantifier, so it can be expressed as a function of x. Thus, by substituting $\psi_{A \leftsquigarrow A'}\ x'$ for x in p by a meta-level operation, we can apply this proof to the hypothesis instantiated earlier with this same inverse embedding value, yielding a proof of $B'[x := \psi_{A \leftsquigarrow A'}\ x']$. However, we still have to prove B'', *i. e.*, $B'[\phi_{A \rightsquigarrow A'}\ x := x']$ by definition.

To unify these two types, we can note that as the substitution performed in the first type replaces the occurrences of x with the inverse embedding of x', it also replaces the occurrences of the embedding of x by a composition of embeddings applied to x':

$$x := \psi_{A \leftsquigarrow A'}\ x' \quad \Longrightarrow \quad \phi_{A \rightsquigarrow A'} x := (\phi_{A \rightsquigarrow A'} \circ \psi_{A \leftsquigarrow A'})\ x'$$

Thanks to the proof of the embedding condition available for x', we can systematically rewrite this composition into an identity wherever it appears in the proof of $B'[x := \psi_{A \leftsquigarrow A'}\ x']$, and the final proof obtained is indeed B''. The contravariant combinator is therefore as follows:

$$\lambda(H : \Pi(x : A).\, B)(x' : A')(c : P\, x').\, \pi_c^{\text{rw}}\ (p[x := \psi_{A \leftsquigarrow A'}\ x']\ (H\ (\psi_{A \leftsquigarrow A'}\ x')))$$

where π_c^{rw} is the rewrite proof of all compositions $\phi_{A \rightsquigarrow A'} \circ \psi_{A \leftsquigarrow A'}$ into identities in the type of the argument, using the proof for which c is the witness.

Let us take a simple statement over any natural number:

```
forall (n : nat), n * 0 = 0
```

Its preprocessing yields the following statement:

```
forall (n' : Z), n' ⩾ 0 → n' * 0 = 0
```

A contravariant proof gives proof `p : n * 0 = 0 → Z.of_nat n * 0 = 0` on the subterm. The necessary proof to allow goal substitution is the following:

```
H : forall (n : nat), n * 0 = 0
n' : Z
c : n' ⩾ 0
─────────────────────────────
          n' * 0 = 0
```

As explained above, we build a new proof based on `p` by replacing variable `n` with `Z.to_nat n'`, and we apply it to `H (Z.to_nat n')`. We obtain a proof of

```
Z.of_nat (Z.to_nat n') * 0 = 0
```

and proof `c` of the embedding condition on `n'` allows us to rewrite this embedding composition into an identity thanks to a user lemma,[11] and conclude.

11: Value π_{2p} when declaring the partial embedding.

6.2.2 Handling logical connectives

Initially, the algorithm traverses the quantifiers as well as the logical parts of the goal, *i. e.*, the various logical connectives, until it reaches the predicates or relations marking the transition to the subterms specific to a theory. At the level of each connector, an associated proof is applied, enabling one or more recursive calls to be made on the subterms. Since the aim is to generate a proof of implication between both goals, we use morphism properties of implication with respect

to logical connectives. Thus, the construction of a proof of implication by traversing a $K\ P_1 \cdots P_n$ connector can be done from the proofs of implication built on the P_i arguments, in one direction or the other depending on the polarity of the position of each argument for this connector.

The expected morphism lemma for K has the following type:

$$\Pi P_1 \cdots P_n\, P'_1 \cdots P'_n.$$
$$(P_1 \diamond^K_1 P'_1) \to \cdots \to (P_n \diamond^K_n P'_n) \to (K\ P'_1 \cdots P'_n \to K\ P_1 \cdots P_n)$$

$$\text{where } \diamond^K_i := \begin{cases} \to & \text{if position } i \text{ is covariant for connector } K \\ \leftarrow & \text{otherwise} \end{cases}$$

Preprocessing a disjunction `A \/ B` amounts to generating `A'` and `B'` respectively as well as a proof of `A' \/ B' -> A \/ B` from subterms `A` and `B`. In this case, Trakt uses a lemma showing that implication is a morphism for disjunction:

```
Lemma or_impl_morphism : forall (A B A' B' : Prop),
  (A' -> A) -> (B' -> B) -> (A' \/ B' -> A \/ B).
```

This is an instance of the general case above, where $\diamond^{\vee}_i = \to$ for all i and $n = 2$ because all positions are covariant, whereas in the case of an implication, the first argument would be in a contravariant position.

Handling boolean logic If the target logical type is different from the logical type of the inspected connector, then an attempt is made to express all the arguments of the connector in the target type. If this is possible, then we can replace the connector with its associated version by adding an implication lemma between both.

If we target `bool` for logic and the goal contains a disjunction in `Prop`, we attempt to pre-process each argument into a Boolean injected into `Prop`. Where possible, we can apply the following lemma that allows us to use boolean disjunction `||` in the output goal:

```
Lemma orb_or_impl : forall (b₁ b₂ : bool),
  b₁ || b₂ = true -> b₁ = true \/ b₂ = true.
```

By induction, it is thus possible to transfer an entire logical tree from `Prop` to `bool` or *vice versa*, when all the atoms allow it.

Logical atoms The term is traversed until a logical atom is found, *i.e.*, either `True` or `False`, or a predicate declared in Trakt under which terms contained in the theory to be preprocessed can be found. This is the case of logical embeddings defined in § 6.1.2, where the proof given by the user is a logical equivalence from which a proof of implication can be obtained. In the case where the arguments of the predicate are in a type eligible for an embedding, the version of the relation used in the output goal introduces embeddings in front of these subterms, paving the way for the specific preprocessing detailed in the next subsection.

We consider the following goal:

```
forall (x : int), x = x
```

We assume that the user has given a proof of embedding from equality over `int` into boolean equality over `Z`, presented in Example 6.1.1. From this proof, we can get an implication in the desired direction — here, we assume the direction to be covariant:

```
forall (x y : int), Z_of_int x =? Z_of_int y = true -> x = y
```

So, when going through this node, we justify the transition to the target relation — boolean equality — and launch two recursive preprocessing calls on `Z_of_int x`, on either side of the equality.

6.2.3 Theory-specific preprocessing

Once it is under the logical atoms, the aim of Trakt is to embed as many values as possible into the desired target type. The algorithm will therefore inspect each node and exploit the various morphism proofs declared by the user. All unknown values are traversed and left untouched.

Embedding function descent Inspired by `zify`, the preprocessing algorithm of Trakt introduces embedding functions as soon as possible into the initial goal before pushing them towards the leaves of the term. These embedding functions are the trigger for all the rewrites. This is because morphism lemmas are used to replace their left-hand member in the input term with their right-hand member in the output term, and the left-hand member has a leading embedding function where possible. The introduction of an embedding, for example by means of a predicate equivalence proof, allows all the morphism lemmas to be used in a cascade down to the leaves of the tree.

Example 6.2.1
Consider the following goal:

```
forall (x y : int), x * y + x = x * (y + 1)
```

If the user has declared a logical embedding from equality over type `int` to equality in `Prop` over type `Z` for example, then the equivalence proof allows the algorithm to be launched on the two members of the equality preceded by an embedding function. If all the operations have been declared embeddable into their counterparts in `Z`, then the left-hand member will undergo this list of rewritings, pushing all the embedding functions down:

```
Z_of_int (x * y + x)
Z_of_int (x * y) + Z_of_int x
Z_of_int x * Z_of_int y + Z_of_int x
```

All the equality proofs are extended to the logical atom in which they are used, so that they can be composed by transitivity and turned into a single equality proof between the input and output logical atoms.

When the type of one of the arguments of a non-interpreted function is embeddable, in order to preprocess this argument correctly, an embedding identity is inserted in front of this argument.[12] This ensures that the argument is preceded

12: Here, we use the first identity provided by the user, which is unconditionally true for any embedding:

$$\psi \circ \phi \doteqdot \mathrm{id}$$

by an embedding function before being preprocessed by Trakt.

Handling the leaves of the tree At the leaves of the tree, Trakt removes as many embeddings as possible to obtain an output goal expressed entirely in the target type. These leaves are either zero-arity constants or variables.

In the first case, the proof provided with the constant allows erasing the remaining embedding in favour of a new constant in the target type. This is the case in Example 6.1.2 where the zero of type `nat` is embedded into the one of type `Z` using a lemma involving an embedding on the left-hand member.

In the second case, that of a variable $x : T$, if type T can be embedded into T', then the variable is necessarily the argument of a composition of embedding functions $\phi_{T \rightsquigarrow T'}$. It then suffices to replace the term $\phi_{T \rightsquigarrow T'}\ x$ with the output variable $x' : T'$. This substitution is valid, the corresponding proof being performed when the output goal is closed. Indeed, repositioning the new quantifier above a translated open term t', containing embeddings on variable x, amounts to extending a proof of $t' \rightarrow t$, obtained by a recursive call, to the following implication between the quantified types:

$$\Pi x' : T'.\, t'[\phi_{T \rightsquigarrow T'} x := x'] \quad \rightarrow \quad \Pi x : T.\, t$$

Exemple 6.2.2
Going back to the goal of the previous example, descending the embedding functions in the two members of the equality gives the following two terms:

```
Z_of_int x * Z_of_int y + Z_of_int x
Z_of_int x * (Z_of_int y + Z_of_int 1)
```

To complete the preprocessing, we apply the lemma allowing us to rewrite the value `1` in type `int` into its counterpart in `Z`, and we close the term with new quantifiers `x'` and `y'`, to obtain the following final goal:

```
forall (x' y' : Z), x' * y' + x' = x' * (y' + 1)
```

6.2.4 The `trakt` tactic

The algorithm presented above has been implemented in the form of a tactic `trakt`. As soon as we switch to proof mode, we call `trakt` to preprocess the goal, before letting an automated proof tactic finish the proof.

This tactic takes two arguments corresponding to the parameters of the algorithm: the target type for the theory being processed and the type to express logic. Thus, if integers are expressed in `Z` and logic in `Prop`, we will call the tactic with these arguments.

Several of the arithmetic goals presented in this part can be proved using various declarations and the following tactic combination:

```
Proof. trakt Z Prop; lia. Qed.
```

When the goal requires the theory of equality, we can use a tactic calling an SMT solver instead of `lia`, such as the `smt` tactic from SMTCoq or the one of the itauto project.

It is also possible to preprocess a goal just to rewrite its logical part, omitting the first argument. We then switch to a boolean goal by calling `trakt bool`.

In the case of a theory for which the only relevant preprocessing before using an automated proof tool is to exploit the decidability of predicates, there is no need to activate the theory-specific pre-processing features of Trakt. We therefore restrict the tool to a logical preprocessing by removing the first argument.

A priori, the information supplied to Trakt must be terms declared before the proof, as this information is stored in a database that persists after the proof has been completed. However, in certain proofs, it may be relevant for some information to be made known to Trakt, particularly relations. The tool has a feature to cover this case, with a new syntax:

```
trakt T L with rel (R, R', π_R).
```

where `T` and `L` are the target types for theory and logic, and the triplet corresponds to a relation declaration as presented in § 6.1.2.

The most telling example to illustrate the need for this feature is decidability of a relation on a local type. Indeed, if a goal quantifies over a type A and there is information implying, for instance, that equality over A is decidable, then it may be relevant to introduce it into the proof context, make the decidability proof explicit, and call `trakt` with this additional piece of information, in order to preprocess the rest of the goal and perhaps obtain a proof that is simpler to complete.

Finally, as the algorithm implemented in the `trakt` tactic allows reasoning in both directions thanks to the polarity management described previously, it can be used for forward chaining preprocessing without any additional effort. We can therefore preprocess a hypothesis rather than the goal, this time by generating a proof of implication of the new hypothesis from the old one. The syntax proposed by Trakt in this case is the following:

```
trakt_pose T L : H as H'.
```

where `H` is the hypothesis to be preprocessed and `H'` the name to be used for the new hypothesis obtained. The other features of Trakt are also available in this direction.

Conclusion and perspectives 7

The TRAKT plugin, whose concepts were presented in the previous chapter, has been implemented in COQ.[1] This chapter evaluates this tool that completes an ecosystem of automation tools available to the COQ user (§ 7.1) and can interact with some of them. TRAKT improves on `zify`'s answer to the problem of preprocessing by canonisation for statements in the SMT class (§ 7.2). The plugin is partially aligned with the specification defined in § 5, but it also has some flaws and could be improved (§ 7.3).

7.1 Ecosystem of automation tools for COQ 61
7.1.1 The need for preprocessing . . 61
7.1.2 Modular transformations of the scope tactic 63
7.2 Success of the plugin 64
7.2.1 Examples of goals handled . . 65
7.2.2 Integration of TRAKT with other tools 66
7.3 Paths of improvement 67
7.3.1 Polymorphism and dependent types 67
7.3.2 Architecture of the preprocessing phase 68

1: The questions related to this implementation are dealt with in § 12.

7.1 Ecosystem of automation tools for COQ

Without automation, software as complex as a proof assistant cannot be used properly. There are an increasing number of assistance tools to help the user in the proofs. Thus, TRAKT fits into an ecosystem of tools used either to automatically prove goals or to preprocess them so that other automation tools work even better. In this section, we outline the difficulty of performing proofs in COQ and the need for preprocessing, citing a few proof automation tools that TRAKT can work with; then we quickly introduce `scope`, another preprocessing tactic that can associate with TRAKT.

7.1.1 The need for preprocessing

Technical yet uninteresting proof steps are the daily bread of program verification. Fortunately, many elementary statements are easily solved by modern automated provers. The corresponding *formal* proof steps can in turn be automated, *e. g.*, using *hammers*, a powerful architecture for connecting external automated theorem provers with formal interactive proof environments. For instance, the COQHAMMER [40] plugin equips COQ with an instance of hammer, providing a tactic called `hammer` that combines heuristics with calls to external provers for first-order logic, so as to obtain hopefully sufficient hints, including relevant lemmas from the current context, to prove the goal. The actual formal proof is then reconstructed from these hints thanks to variants of the `sauto` tactic and `hammer` outputs a corresponding robust and oracle-independent proof script.

[40]: CZAJKA *et al.* (2018), "Hammer for Coq: Automation for dependent type theory"

For example, consider a property on the length of the reversed concatenation of two lists:

```
Lemma length_rev_app : forall (B : Type) (l l' : list B),
  length (rev (l ++ l')) = length l + length l'.
```

This lemma can be proved with COQHAMMER, which provides the following script, using auxiliary lemmas `app_length` and `rev_app_length` from the section of the standard library dealing with lists:

```
Proof. scongruence use: app_length, rev_length. Qed.
```

Yet, as of version 1.3.2, COQHAMMER is not designed to exploit any theory-specific reasoning, and thus cannot prove this slight variant, where `b :: l'` replaces `l'`, because it lacks arithmetical features:

```
Lemma length_rev_app_cons : forall (B : Type) (l l' : list B) (b : B),
  length (rev (l ++ (b :: l'))) = length l + length l' + 1.
```

In this case, users may resort to the SMTCoq plugin, implementing a certificate checker for proof witnesses output by SMT solvers. The latter automated provers are indeed tailored for finding proofs combining propositional reasoning, congruence and theory-specific decision procedures, *e.g.*, for linear arithmetic. However, none of CoqHammer or SMTCoq can in general reason by case analysis or induction.

A variant of the `sauto` tactic can very well prove the first goal below about list concatenation, but not the second one in which the first list is reversed:

```
Lemma app_nilI : forall (B : Type) (l l' : list B),
  l ++ l' = [] → l = [] ∧ l' = [].

Lemma app_nil_rev : forall (B : Type) (l l' : list B),
  rev l ++ l' = [] → l = [] ∧ l' = [].
```

The SMTCoq plugin can be used to prove properties of linear integer arithmetic, but only when they are stated using the type `Z` of integers from Coq's standard library:

```
Lemma eZ : forall (z : Z), z ≥ 0 → z < 1 → z = 0.
```

Up to version 2.0, SMTCoq is however clueless about any alternative instance of integer arithmetic, *e.g.*, the type `int` of unary integers from the MathComp library, already mentioned previously.

```
Lemma eint : forall (z : int), z ≥ 0 → z < 1 → z = 0.
```

Fortunately, Coq distributes the `lia` tactic, specific to linear integer arithmetic, that can actually also prove lemmas such as `eZ`. Moreover, `lia` can be customised to a user-defined instance of arithmetic thanks to the `zify` dedicated preprocessing, presented in § 5.2. Once correctly configured for type `int` thanks to the `mczify` extension, `lia` is equally powerful on type `int` or type `Z` and proves both `eZ` and `eint`. However, as powerful as it may be on integer linear arithmetic, the tactic is by nature unaware of the theory of equality. Hence, although it can prove equality `eintC` from below, it is unable to prove the variant `cong_eintC`, because the latter involves a congruence with the `_ :: nil` operation, alien to the theory of linear integer arithmetic.

```
Lemma eintC : forall (z : int), z + 1 = 1 + z.
Lemma cong_eintC : forall (z : int), (z + 1) :: nil = (1 + z) :: nil.
```

Proving the property expressed by `cong_eintC` requires *combining* different theories, in this case integer arithmetic and the theory of equality, as SMT solvers do. Yet, in this case as well, the SMTCoq plugin cannot help, because the statement of this fact is phrased using type `int` instead of `Z`. The recent itauto SAT solver [55], implemented in Coq, provides an alternate take on formally verified satisfiability modulo theory, and organises the cooperation between the independent tactics `lia`, for integer arithmetic, and `congruence`, for equality. As a consequence, the `smt` tactic built on top of itauto can benefit from `lia`'s preprocessing facilities. For instance, as soon as `lia`'s preprocessing is correctly configured for type `int`, the `smt` tactic is able to prove lemma `cong_eintC`.

[55]: Besson (2021), "Itauto: An Extensible Intuitionistic SAT Solver"

However, `lia`'s preprocessing facilities are not known to the rest of the SMT decision procedure. Thus, although the first goal below is solved by the latter `smt` tactic, because `lia` has been informed of the boolean equality test == available

on type int, the same tactic fails on the cong_eintCb variant, featuring an uninterpreted symbol f:

```
Lemma eintCb : forall (z : int), (z + 1 =? 1 + z) = true.
Lemma cong_eintCb : forall (f : int → int) (z : int),
  (f (z + 1) =? f (1 + z)) = true.
```

As it turns out, although a variety of tactics implementing automated reasoning is available to the users of the Coq proof assistant, finding the appropriate weapon for attacking a given goal remains challenging. It is often quite difficult to anticipate the exact competence of tactics based on first-order automated reasoning, and to interpret failure. As a consequence, large-scale formalisation endeavours may end up developing their own specific automation tools, like the list_solve tactic in the Verified Software Toolchain [56, 57], for automating reasoning about lists and arithmetic, which makes the number of available tactics multiply even more, often redundantly. A form of generic preprocessing such as that targeted by Trakt therefore seems ideal to make existing automation tools more flexible.

[56]: Appel (2011), "Verified Software Toolchain - (Invited Talk)"

[57]: Appel *et al.* (2022), *Verifiable C*

7.1.2 Modular transformations of the scope tactic

The scope [12] tactic is a combination of various preprocessing tactics with the objective of reducing the Coq statements belonging to the SMT fragment to the logical theory handled by SMT solvers, living at a lower level.[2] The preprocessing performed by scope is orthogonal and complementary to the features of Trakt, so the two tools can work together in preprocessing SMT goals. Here, we present three of the transformations performed by scope: generation of the inversion principle for inductive relations, pattern matching elimination, and hypothesis monomorphisation.

[12]: Blot *et al.* (2023), "Compositional preprocessing for automated reasoning in dependent type theory"

2: For example, quantifiers are in prenex positions and there are no dependent types.

Inversion principle for inductive relations An inductive relation is an inductive type representing a relation between terms, whose codomain is often Prop. When a hypothesis is a witness of the relation between terms, it is possible to determine from these terms the various constructors that may have been used to build the witness. The inversion tactic uses this property to add hypotheses to the context, but when using an SMT solver, it can be interesting to keep the inversion principle as an additional hypothesis. The associated transformation in scope does this job.

We consider the inductive relation representing the graph of the addition function between two natural numbers:

```
Inductive add : nat → nat → nat → Prop :=
  | add0 : forall (n : nat), add 0 n n
  | addS : forall (n m k : nat), add n m k → add (S n) m (S k).
```

A call to this transformation tactic adds a new hypothesis to the context, having the following type:

```
forall (n m k : nat), add n m k ↔
  (exists (n' : nat), n = 0 ∧ m = n' ∧ k = n') ∨
  (exists (n' m' k' : nat),
    add n' m' k' ∧ n = S n' ∧ m = m' ∧ k = S k')
```

Pattern matching elimination Pattern matching, available in a high-level language, is an unknown construct in an SMT solver. When a hypothesis contains pattern matching, this transformation splits it into as many new hypotheses as the number of cases.

Consider accessing the n-th element of a list. This function can be written in a total way, either by using an optional return value to take into account the case where n is greater than the size of the list, or by using a default return value. A hypothesis giving the definition of the latter (`nth_default`) from the former (`nth`):

```
forall (A : Type) (d : A) (l : list A) (n : nat),
  nth_default A d l n =
    match nth l n with
    | Some x ⇒ x
    | None ⇒ d
    end
```

is replaced with two hypotheses, each one focusing on a case of the pattern matching:

```
H1 : forall A d l n, nth l n = Some x → nth_default d l n = x
H2 : forall A d l n, nth l n = None   → nth_default d l n = d
```

Monomorphisation Most automated theorem provers do not handle polymorphism. However, many lemmas in Coq are polymorphic. It is therefore useful to implement a transformation that instantiates polymorphic hypotheses with the types present in the goal, so that the solver can exploit them. The instantiation of lemmas is performed by a heuristic that selects various types appearing in the goal as potentially interesting instances.

In the following proof context, by instantiating `H` with `option Z` and `list unit`, the proof becomes trivial for an SMT solver.

```
H : forall (A B : Type) (x1 x2 : A) (y1 y2 : B),
       (x1, y1) = (x2, y2) → x1 = x2 ∧ y1 = y2
─────────────────────────────────────────────────
        Z.of_nat n + Z.of_nat n ⩾ Z.of_nat n
```

7.2 Success of the plugin

Given the lack of existing tools with the purpose of canonising statements as a bridge between the expressiveness of Coq and the various decision procedures, the presence of Trakt improves the level of automation of several tactics available to Coq users. The use of Trakt does not show any notable regression compared with `zify`, its main point of comparison, with regard to the specification identified in § 5.1. Notations linked to *ad hoc* polymorphism such as those of MathComp presented in Example 5.2.3 are supported by Trakt, offering a comparable feature. Finally, the dedicated preprocessing for logic in Trakt improves the situation for the SMTCoq plugin, which was the original aim while designing this tool. This section uses a few example goals to demonstrate the effective features of Trakt, and then shows how the plugin can be used efficiently with SMTCoq.

7.2.1 Examples of goals handled

In the previous chapter, examples were chosen to illustrate precise aspects of how Trakt works. Here, let us take a concrete example and detail the entire preprocessing phase from the user's point of view.

The first example goal is the following:

```
forall (x : int), x * x ⩾ 0
```

By displaying MathComp generic projections, the full goal is the following:

```
forall (x : int),
  @Order.ge int_porderType
    (@GRing.mul int_Ring x x) (@GRing.zero int_Ring) = true
```

First of all, we need to determine the canonical type of integers for the automation tactic under consideration, as well as the ideal logical type for this tactic to work best. In this particular case, we can use the `lia` tactic and target the type `Z` for integers and `Prop` for logic, for example.

An embedding must therefore be declared between `int` and `Z`. Next, the operations used in the goal must also be declared. Here, the operation is multiplication `mulz`, whose counterpart in `Z` is `Z.mul`. Non-negative constants of type `int`, in particular zero, are represented as embeddings from `nat` to `int` *via* the function `Posz`. If we declare an embedding from `nat` to `Z`, then it suffices to show that `Posz` can be embedded into identity in `Z` so that all constant values of type `int` can be embedded into `Z` with Trakt. The order relation over `int` must also be declared embeddable into `Z.ge`. Finally, the generic projections of MathComp must be declared as conversion keys. Here is the equivalent using the Trakt commands:

```
Trakt Add Embedding int Z Z_of_int Z_to_int π₁ π₂.
Trakt Add Symbol mulz Z.mul π₃.
Trakt Add Embedding nat Z Z.of_nat Z.to_nat π₄ π₅.
Trakt Add Symbol Posz (@id Z) π₆.
Trakt Add Relation (@Order.ge int_porderType) Z.ge π₇.
Trakt Add Conversion GRing.mul.
Trakt Add Conversion GRing.zero.
Trakt Add Conversion Order.ge.
```

The proofs used have the following types:

```
π₁ : forall (x : int), Z_to_int (Z_of_int x) = x
π₂ : forall (x' : Z), Z_of_int (Z_to_int x') = x'
π₃ : forall (x y : int), Z_of_int (x * y) = Z_of_int x * Z_of_int y
π₄ : forall (n : nat), Z.to_nat (Z.of_nat n) = n
π₅ : forall (n' : Z), n' ⩾ 0 → Z.of_nat (Z.to_nat n') = n'
π₆ : forall (n : nat), Z_of_int (Posz n) = Z.of_nat n
π₇ : forall (x y : int), x ⩾ y ↔ Z_of_int x ⩾ Z_of_int y
```

We can then call `trakt Z Prop` and get the following goal, provable by the `lia` tactic, as opposed to the initial goal without preprocessing:

```
forall (x' : Z), x' * x' ⩾ 0
```

Let us now show how two other goals we encountered earlier can be proved using preprocessing by Trakt.

Example 7.2.1
We take the goal of Example 5.2.4:

```
forall (f : int → int) (x : int), f (2 * x) ≤? f (x + x)
```

Subject to declarations similar to the previous example, preprocessing towards `Z` and boolean logic with `trakt Z bool` yields the following goal, provable by a tactic delegating proofs to an SMT solver:

```
forall (f' : Z → Z) (x' : Z), f' (2 * x') ≤? f' (x' + x') = true
```

Example 7.2.2
Now, we take the goal of Example 6.2.1, replacing `int` with `nat`:

```
forall (x y : nat), x * y + x = x * (y + 1)
```

Preprocessing with `trakt Z Prop` gives the following goal, provable by `lia`:

```
forall (x' : Z), x' ⩾ 0 → forall (y' : Z), y' ⩾ 0 →
  x' * y' + x' = x' * (y' + 1)
```

It can be seen from these examples that combining Trakt with existing automation tactics effectively extends their input domain by making more signatures intelligible.

7.2.2 Integration of Trakt with other tools

Trakt can be used as a one-off preprocessing tool before using an automated theorem prover, but it can also be used with several other preprocessing tools. For example, Trakt has been integrated into the `scope` suite of transformations presented in § 7.1.2, then used in association with SMTCoq, giving the Sniper plugin [12]. The role of Trakt in this context is to canonise arithmetic and to ensure that logic is expressed in `bool` as much as possible, in order to exploit decidability of the predicates present in the goal. Here we look at an example of a highly automated formalisation using Sniper.

[12]: Blot *et al.* (2023), "Compositional preprocessing for automated reasoning in dependent type theory"

Example 7.2.3
This example deals with a formalisation of properties of several variants of λ-calculus, such as strong normalisation, based on the MathComp library. This formalisation[3] includes deep embeddings of languages with binders, in which De Bruijn indices are used to represent bound variables. The price to pay is the need to prove technical and uninteresting properties about variable substitution and shifting. Goals in such a context often contain both arithmetic and logical reasoning, and proofs require inductive reasoning. For instance, untyped λ-calculus is defined as:

3: We owe this example to Kazuhiko Sakaguchi.

```
Inductive term : Type :=
  | var of nat
  | app of term * term
  | abs of term.
```

with the following shift and substitution functions:

```
Fixpoint shift d c t : term :=
  match t with
  | var n ⇒ var (if c ≤ n then n + d else n)
```

Term `shift d c t` is term `t` in which variables with an index above threshold `c` have been shifted by `d` positions.

Term `subst n ts t` is term `t` in which variables with an index above threshold `n` have been replaced with terms contained in list `ts`.

```
  | app t1 t2 ⇒ app (shift d c t1) (shift d c t2)
  | abs t1 ⇒ abs (shift d c.+1 t1)
  end.

Notation substv ts m n :=
  (shift n 0 (nth (var (m - n - size ts)) ts (m - n))).
Fixpoint subst n ts t : term :=
  match t with
  | var m ⇒ if n ⩽ m then substv ts m n else m
  | app t1 t2 ⇒ app (subst n ts t1) (subst n ts t2)
  | abs t' ⇒ abs (subst n.+1 ts t')
  end.
```

Note that these definitions use addition, subtraction, and comparison on the natural numbers defined in MathComp. By adding them to the Trakt database, then performing an induction on the terms of the calculus followed by a call to `snipe`, the main tactic of the Sniper project, we can automatically prove a number of properties on this λ-calculus.

```
Lemma shift_add d d' c c' t :
  c ⩽? c' → c' ⩽? c + d →
    shift d' c' (shift d c t) = shift (d' + d) c t.
Proof. revert d d' c c'; induction t; snipe. Qed.

Lemma shift_shift_distr d c d' c' t :
  c' ⩽? c →
    shift d' c' (shift d c t) = shift d (d' + c) (shift d' c' t).
Proof. revert d d' c c'; induction t; snipe. Qed.
```

7.3 Paths of improvement

Although Trakt finds a use in the ecosystem of preprocessing tools for Coq, the tool remains limited in some aspects. This section outlines these limitations to justify further work.

7.3.1 Polymorphism and dependent types

Despite various efforts to make the translation more general, as the starting point for this thesis, Trakt was designed with `zify` as a template. As such, the plugin inherits the various *ad hoc* aspects of the latter, being intended to preprocess arithmetic provable by `lia`. In particular, embeddings follow the type class model of `zify`. Trakt certainly brings a level of flexibility to it due to the fact that the storage of terms is external — at the meta level — unlike type class instances that must strictly conform to a Coq type, but by design, the class of terms that can be declared is close to that of `zify`.

In fact, the type embeddings defined in § 6.1.1 concern simple types expressed in a single term, which can be limiting in the case where we wish to declare polymorphic embeddings or embeddings based on dependent types. We can cite the example of ordinal numbers, bounded natural numbers that are always partially embeddable into `nat` or a larger integer type such as `Z`, the embedding condition being respecting the upper bound of the source ordinal.

Another interesting example is canonisation when using container types on integers. The following goal:

```
forall (l : list int), sum_int l ⩾ 0
```

could be rewritten in Z for what concerns arithmetic, without affecting the rest. Indeed, if $\texttt{sum}_{\texttt{int}}$ is defined as `fold_left (+) 0 l`, then we can rewrite the addition and the zero as those of Z to obtain the target sum. However, this kind of embedding using polymorphic types is not allowed by Trakt.

A second problem is the impossibility of not processing a value. In fact, in goals involving dependent types that we wish to leave unchanged in the output goal, some values can be used as arguments whose type must not change during translation, even though it is an embeddable type. One example is bitvectors whose integer size must remain within the type initially used to encode it, because embedding this value into another type introduces the risk of getting an ill-typed output goal. Trakt does not handle these special cases and always performs the embedding when it is possible.

7.3.2 Architecture of the preprocessing phase

The structure of the preprocessing algorithm also limits the possibilities of Trakt. Indeed, the tool was designed in an incremental way. Initially built on the principle of composing morphism proofs to preprocess subterms belonging to a theory whose signature can be embedded, the various additional features were grafted onto the algorithm's main recursive function, adding arguments to represent various pieces of information to be held in memory in the recursive calls. This approach has the advantage of being pragmatic and quickly providing a functional tool that can be used in a real context, but it also has a few drawbacks.

Firstly, leaving aside congruence theory, that can always be processed on the fly, Trakt only allows one theory to be preprocessed at a time. If the goal contains a mix of theories, Trakt has to be called several times with preprocessing for one theory each time.

Secondly, going from `bool` to `Prop` is done in two phases. In order to rewrite a boolean subterm in `Prop`, the subterm `b` must be cast in `Prop` with an equality: `b = true`, `false = b`, *etc.* If the subterm is under an uninterpreted predicate of type `bool → Prop`, then it will not be possible to express it in `Prop` in the output goal. In the current state of Trakt, this information is not tracked during translation. When translating a boolean value, it is therefore not possible to know whether it can be replaced with its counterpart in `Prop`. Even if the boolean equality `Nat.eqb` over natural numbers can be rewritten into the equality in `Prop` over type Z, at the node of the relation, it will not be possible to know if the embedding can be carried out. So, to translate from `bool` to `Prop`, a specialised logical phase is run first, with the ability to look over a term to see if it is cast into `Prop` in a way that allows rewriting. Once the first pass has been made, the goal is expressed in `Prop` and the remaining embeddings are possible. We will therefore go from `Nat.eqb` to `@eq nat` and then to `@eq Z`, requiring two user declarations instead of one.

Finally, rewriting proofs take the form of an equality between a subterm before and after rewriting. Their composition is done by transitivity, so all the equality proofs are extended to have the same context and to be able to be composed. As a result, the context is repeated with a slight variation at each rewrite, which gives

the global proof on a subterm a quadratic complexity in space as a function of the number of rewrites to be performed in this subterm.

All in all, the flaws identified in this section can be corrected with *ad hoc* meta-programming solutions, which allows keeping the current prototype. Alternatively, drawing on the lessons learnt from the design of Trakt, we can also take up the problem of proof transfer, presented in § 3.3, and design a new preprocessing tool with a more general approach, encompassing cases lying at the boundary of what Trakt can handle. This is the solution chosen for the second prototype developed during this thesis, presented in the following part.

Trocq:

Proof transfer

by parametricity

Introduction

The limitations identified while reviewing Trakt show a need for generalisation compared to the *ad hoc* approach inherited from `zify`. Rather than starting from concrete relations between terms[4] and building around them an algorithm that exploits the embedding functions to create the associated goal, we can abstract these relations, first build the algorithm that exploits them, and then give them content. We then say that two types A and B are related if there is a relation R of type $A \to B \to \Box$. Next, we study the possibility to propagate these relations by induction on typing.

4: In Trakt, these are, for example, bijections or partial embeddings defined in § 6.1.1 to relate types, or proofs of morphism with respect to the embedding function defined in § 6.1.3 to link operations.

This study is the subject of a line of work on the concept of *parametricity* or *logical relations* [58], the aim of which was initially to derive properties on terms from their types in polymorphic λ-calculi. By giving types a relational interpretation, we can obtain so-called "free" theorems. If two terms a and b are related, then we can find a relation between two terms $C[a]$ and $C'[b]$, where C is a context and C' is the associated context. This means, for example, that a relation between two types A and B can be extended to lists of values in A and B, which in practice corresponds to operations like `List.for_all2` in OCaml. The extension of parametricity to dependent types and then to the type theory of Coq makes it possible to internalise parametricity witnesses, *i. e.*, proofs that two terms are linked by a relation. In such a context, free theorems actually become Coq proofs, rather than meta-level properties as before.

[58]: Mitchell (1986), "Representation Independence and Data Abstraction"

The implementation of a parametricity translation [59] is a function that takes a term to be translated — in our case, the goal — and produces two output terms, a translated term — the associated goal — and a witness proving that the original term is related to the translated term. In the empty context and on closed terms, the parametricity translation in Coq is nothing other than a deep identity on the goal, obtained by traversing it by induction on the syntax. To translate constants, we add relations between each source constant c and an associated target constant c'. So, to translate addition in `nat` to addition in `bin_nat`, we provide a relation R between both types as well as a relation between both additions, *i. e.*, a proof that the addition of terms related by R yields terms related by R. Any goal containing values and additions in `nat` can then be translated to an associated goal mentioning these terms in `bin_nat`.

[59]: Boulier *et al.* (2017), "The next 700 syntactical models of type theory"

In order to carry out a proof transfer, we need to be able to extract a function from the parametricity witness. We then wish to enrich the relation propagated through the syntax during the parametricity translation, in order to take advantage of the general framework that this technique provides while obtaining more information in the parametricity witness obtained at the end of the translation. Of all the possible enrichments, asymmetrical enrichments cannot be propagated through all constructs, for reasons of polarity, particularly for the Π-type. Stable enrichment is possible through so-called *univalent* parametricity [14], at the cost of adding the *univalence* principle to Coq, which is necessary to translate universes. The witness obtained is then dense and rich in information, and contains in particular the function necessary for proof transfer.

[14]: Tabareau *et al.* (2021), "The marriage of univalence and parametricity"

It is therefore possible to carry out proof transfer using a parametricity translation. Such a translation supports all language constructs and allows many goals to be preprocessed. However, in this context, relations on constants added to the context by the user before translation require a univalent witness that, due to the

richness of this witness, is not always easily provable. Furthermore, the principle of univalence comes in the standard version of Coq in the form of an axiom that is regrettable to use as part of the preprocessing of a goal that could be done without an axiom if the user did it by hand, by making manipulations similar to what is done automatically by Trakt. This limitation is the major motivation for the development of Trocq [15], an implementation of a new, more flexible and modular parametricity framework, in order to retain the generality of parametricity while making parsimonious use of axioms, ideally reduced to cases where they are strictly necessary to process the input term.

[15]: Cohen *et al.* (2024), "Trocq: Proof Transfer for Free, With or Without Univalence"

This section presents the theoretical concepts of Trocq. First, we present in more details the context of this new plugin,[5] *i.e.*, the path from the origins of parametricity to univalent parametricity, that forms the basis of the work done on Trocq, in the same way as `zify` for Trakt (§ 8). Secondly, we study the decomposition of the univalent parametericity witness to expose its information, as well as its recomposition into a *hierarchy* of parametricity witnesses and the construction of a single framework that makes all these witnesses work together (§ 9). Finally, we formulate this relation as a logical program, in order to expose as much of the required context information as possible, with the aim of implementing it as a translation in a tactic later on (§ 10).

5: The implementation is available in the repository:

https://github.com/coq-community/trocq

Parametricity in dependent type theory

8

The limitations of TRAKT lead us to look for a more general approach to proof transfer in COQ. This chapter presents *parametricity*, a concept from the theory of programming languages that gives the types of the λ-calculus a relational interpretation, allowing the redesign of the links between input and output terms in the context of a preprocessing tool. Here, we define the original concept and its extension to the λ-calculus of COQ (§ 8.1). Then, we present *univalent parametricity*, an enriched version allowing generalised proof transfer in COQ thanks to the univalence axiom, and we show why this technique is interesting regarding the specification we gave for TRAKT initially (§ 8.2).

8.1 Motivation and definition . . 73
8.1.1 Typing and properties of λ-terms 73
8.1.2 Raw parametricity translation 74
8.1.3 Limitations of the raw translation 75
8.2 Univalent parametricity . . . 75
8.2.1 Enrichment of parametricity witnesses 76
8.2.2 Type equivalence and univalence 77
8.2.3 Univalent parametricity translation 79
8.2.4 Omnipresence of the univalence axiom 81

8.1 Motivation and definition

The notion of *parametricity* dates back to the appearance of polymorphism in λ-calculus. Originally a tool for reasoning about the properties of polymorphic functions [13], it is now used to perform translations of terms. This section presents these two aspects.

[13]: REYNOLDS (1983), "Types, Abstraction and Parametric Polymorphism"

8.1.1 Typing and properties of λ-terms

In the simply typed λ-calculus, terms can have simple functional types or "arrow types", governed by the following typing rule $\text{LAM}_{\rightarrow}$:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \rightarrow B}$$

All functions of this calculus are *monomorphic* and have a concrete domain A and codomain B. Thus, by defining a type $\mathbb{B}$ and primitive operations $\neg$ and $\wedge$ to represent booleans and logical connectives of negation and conjunction, the function representing the NAND logical gate has a unique type:

$$\Gamma \vdash \lambda b_1\, b_2.\, \neg\, (b_1 \wedge b_2) : \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}$$

However, in a λ-calculus *à la* CURRY,[1] there are functions with several possible types. A primitive example is the identity function, $\lambda x.\, x$. Since it can be lawfully applied to any term in the language, this function has an infinite number of types, each one depending on the type chosen for the bound variable: the function is said to be *polymorphic*. When this type can be abstracted and represented by a variable, we refer to *parametric polymorphism*. System F allows representing these variables using an additional binder, Λ. In this way, identity can be uniquely typed:

1: This is the presentation in which functions have no typing annotations.

$$\Gamma \vdash \lambda x.\, x : \Lambda \alpha.\, \alpha \rightarrow \alpha$$

The main observation of *parametricity* is that the body of a polymorphic function never exploits the type of the bound variable, *i. e.*, the *type parameter*. This makes it possible to extract properties about the behaviour of polymorphic functions from their type, *i. e.*, without needing to inspect the body of the function. For

instance, the polymorphic type of the identity function can be used to uniquely determine its implementation. Indeed, the function receives as input a value of a type whose structure it cannot exploit, because it is polymorphic, and must return a value of the same type. As a result, there is no possible implementation other than the one that returns the input value unchanged. These properties, available directly at the type level, are referred to as "free theorems" in the literature [60].

[60]: Wadler (1989), "Theorems for Free!"

8.1.2 Raw parametricity translation

Type theory allows representing parametricity properties and their proofs *internally*, *i. e.*, in the same calculus as the one which the terms being studied live in. In this way, results previously obtained by manual analysis at the meta level can in this context be given automatically by a *syntactic translation* from the calculus to itself. Such translations can take into account dependent types [61], inductive types [62], as well as the full Calculus of Inductive Constructions,[2] including its impredicative sort [63].

[61]: Bernardy *et al.* (2011), "Realizability and Parametricity in Pure Type Systems"

[62]: Bernardy *et al.* (2012), "Proofs for free - Parametricity for dependent types"

2: It is the Calculus of Constructions, defined in § 2.1.3, equipped with inductive types defined in § 2.2.2.

[63]: Keller *et al.* (2012), "Parametricity in an Impredicative Sort"

The work of Bernardy *et al.*, Keller, and Lasson makes it possible to define what will from now be referred to as the *raw parametricity* translation, that essentially defines a logical relation $[\![T]\!]$ for any type T, by induction on the syntax:

$$
\begin{aligned}
[\![\langle\rangle]\!] &:= \langle\rangle \\
[\![\Gamma, x : A]\!] &:= [\![\Gamma]\!], x : A, x' : A', x_R : [\![A]\!]\, x\, x'
\end{aligned}
$$

$$
\begin{aligned}
[\![\Box_i]\!] &:= \lambda A\, A'.\, A \to A' \to \Box_i \\
[\![x]\!] &:= x_R \\
[\![\Pi x : A.\, B]\!] &:= \lambda f\, f'.\, \Pi(x : A)(x' : A')(x_R : [\![A]\!]\, x\, x').\, [\![B]\!]\, (f\, x)\, (f'\, x') \\
[\![\lambda x : A.\, t]\!] &:= \lambda(x : A)(x' : A')(x_R : [\![A]\!]\, x\, x').\, [\![t]\!] \\
[\![t\, u]\!] &:= [\![t]\!]\, u\, u'\, [\![u]\!]
\end{aligned}
$$

Figure 8.1: Raw parametricity translation for CC_ω.

This presentation uses the standard convention that t' is the term obtained from a term t by replacing every variable x in t with a fresh variable x'. A variable x is translated into a variable x_R with a fresh name. This translation preserves typing in the following sense:

Theorem 8.1.1 (Abstraction theorem)
If $\Gamma \vdash t : T$, then $[\![\Gamma]\!] \vdash t : T$, $[\![\Gamma]\!] \vdash t' : T'$, and $[\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!]\, t\, t'$.

Proof. See for example [63]. ∎

This translation generates precisely the statements expected for a family of types or a parametric program. For instance, the translation of a dependent product given above is a relation that links two functions f and f' if they produce related terms when given related terms as input.

8.1.3 Limitations of the raw translation

In our context, raw parametricity has two main limitations, namely, the fact definitional equalities are not preserved and the weakness of parametricity witnesses.

In the first case, TABAREAU *et al.* take the example of a proof of false[3] from a contradictory equality over natural numbers:

3: That is, an empty inductive type called `False`.

```
forall (n : nat), 0 = S n → False
```

We can prove this property by defining a type constructor `P : nat → Type` that is `0 = 0` in `0` and `False` otherwise. Next, we introduce `n` and the equality `e`, and we must prove `False`. The proof is then obtained by dependent induction on `e`: if we can prove a property `Q 0 refl`, then we have `Q (S n) e`. By posing `Q n _ := P n`, we must then prove `P 0` to obtain a proof of `P (S n)`. Since `P 0` is defined as `0 = 0`, it is sufficient to provide `refl` for this proof, then we obtain the proof of `P (S n)`, *i. e.*, `False`. However, in this last step, we exploit the fact that the value `P` is defined using the induction principle on `nat`, itself defined using pattern matching on the value of type `nat` supplied to it. So, as soon as the head constructor of the argument of `P` is known, we can carry out a ι-reduction step[4] to select the right branch of the pattern matching. This conversion step is necessary to conclude that the final term is well typed. We say that `P 0` and `P (S n)` are *definitionally* equal to — respectively — `0 = 0` and `False`.

4: This is the reduction rule dealing with pattern matching.

If we use raw parametricity to translate the final proof term, we will associate each constant on `nat` with a constant on a type associated with `nat`, for example `bin_nat`. In particular, the induction principle of `nat` will become an induction principle on `bin_nat` whose type structure is the same, *i. e.*, induction takes place from successor to successor as on type `nat`, whereas the constructors of `bin_nat` encode binary values. Consequently, the induction principle that we associate with `nat_rect` is not, unlike the latter, defined by directly pattern matching on its argument. Thus, the translated proof term will not be able to exploit ι-reduction in typechecking. Indeed, Coq will have to show that `refl` has type `P' b0`, which is impossible without the information that the latter term is actually `b0 = b0`. The use of conversion in typing is powerful, but penalises all translations that do not preserve definitional equalities, as is the case with raw parametricity.

The second limitation of this translation is the weakness of the parametricity witnesses. This is because, although the raw translation is able to generate the desired goal after preprocessing,[5] in this context, the parametricity witnesses relating the input and output terms are always relations or relation witnesses. Generating the associated goal is part of the work required for proof transfer, but just knowing that both goals are related is not enough to replace the first with the second. This is because rewriting the goal requires a function from one to the other, which is not provided by the raw parametricity translation.

5: Provided that the parametricity context contains the various relations describing the substitutions desired by the user.

8.2 Univalent parametricity

To overcome the limitations of raw parametricity, one solution is to enrich parametricity witnesses so that it is still possible to get an implication between the output goal and the input goal after translation. This is the promise of *univalent parametricity* [14], a more powerful parametricity translation based on exploiting the *univalence axiom* [64] and user declarations of equivalences between

[14]: TABAREAU *et al.* (2021), “The marriage of univalence and parametricity”

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

types. In this section, we give details about the progressive enrichment of the parametricity witness type, we make explicit the definition of equivalence used in this context, then we introduce the univalent parametricity translation.

8.2.1 Enrichment of parametricity witnesses

To define a parametricity translation, we first define the translation of the universe, as this gives the structure of all parametricity witnesses relating types. We then define the rest of the cases in the translation that extend proofs on subterms to new constructions. For example, in the raw parametricity translation, the witness relating two types A and B is a relation $A \to B \to \square$, and the translation of the dependent product corresponds to a proof of the following property, that states that from witnesses on A and A' and B and B', we can build a witness on dependent products $\Pi a : A.\, B\, a$ and $\Pi a' : A'.\, B'\, a'$:

$$\begin{aligned}
&\Pi(A\, A' : \square)(A_R : A \to A' \to \square).\\
&\Pi(B : A \to \square)(B' : A' \to \square).\\
&\quad (\Pi(a : A)(a' : A').\, A_R\, a\, a' \to (B\, a \to B'\, a' \to \square)) \to\\
&\quad\quad (\Pi a : A.\, B\, a) \to (\Pi a' : A'.\, B'\, a') \to \square
\end{aligned}$$

If we change the definition of the witness on universes to enrich the parametricity translation, the other cases in the translation have more information about the subterms, but they also have to propagate more information. Thus, if we redefine the parametricity witness over universes as a pair $(A \to B \to \square) \times (B \to A)$ containing a function, the property to be proved for an arrow type[6] then becomes the following:

6: We first study the arrow type, then the dependent product, to expose two different problems appearing with the enrichment of the witness by a function.

$$\begin{aligned}
&\Pi(A\, A' : \square)(A_R : (A \to A' \to \square) \times (A' \to A)).\\
&\Pi(B\, B' : \square)(B_R : (B \to B' \to \square) \times (B' \to B)).\\
&\quad ((A \to B) \to (A' \to B') \to \square) \times ((A' \to B') \to (A \to B))
\end{aligned}$$

The left part of the pair to build on the arrow type — the relation — can be obtained in the same way as for the raw translation. The right-hand side, however, amounts to building a value in $A \to B$ from the following three values:

$$f' : A' \to B' \qquad \psi_A : A' \to A \qquad \psi_B : B' \to B$$

Yet, we can see that the contravariance of the domain of the arrow type prevents us from carrying out this proof. In fact, this proof would be feasible if the ψ_A function were in the other direction, *i. e.*, a function ϕ_A of type $A \to A'$. We would build the expected function by taking a value in A, then applying in turn ϕ_A, f', then ψ_B, to obtain a value in B. Breaking the symmetry by only introducing a function in one direction requires us to orient the parametricity witnesses and have a translation that handles two types of witness.

This situation is acceptable, although it complicates the parametricity translation. However, enrichment by a function poses another problem linked to dependent types.[7] The example above deals with the arrow type, but the proof for

7: This problem would also exist when making the witness symmetrical by enriching the raw witness with a function in both directions.

a dependent product is as follows:

$$\begin{aligned}
&\Pi(A\,A' : \square)(A_R : (A \to A' \to \square) \times (A' \to A)).\\
&\Pi(B : A \to \square)(B' : A' \to \square).\\
&\quad (\Pi(a : A)(a' : A').\, A_R.1\, a\, a' \to ((B\,a \to B'\,a' \to \square) \times (B'\,a' \to B\,a)))\\
&\qquad \to ((\Pi a : A.\, B\,a) \to (\Pi a' : A'.\, B'\,a') \to \square)\\
&\qquad\qquad \times ((\Pi a' : A'.\, B'\,a') \to (\Pi a : A.\, B\,a))
\end{aligned}$$

Again, we focus on the right-hand side, where we need to build a value in the dependent type $\Pi a : A.\, B\,a$ from the following three values:

$$\begin{aligned}
&f' : \Pi a' : A'.\, B'\,a'\\
&\psi_A : A' \to A\\
&\psi_B : \Pi(a : A)(a' : A').\, A_R.1\, a\, a' \to (B'\,a' \to B\,a)
\end{aligned}$$

The contravariance certainly gives the wrong type for ψ_A, but here we are interested in the type of ψ_B, that in the case of a dependent product, becomes dependent on two values a and a' as well as a parametricity witness between them. We need to build a value of type $\Pi a : A.\, B\,a$, *i. e.*, a value of type $B\,a$ when introducing a value $a : A$ into the context. However, even if we had a value of type A' to supply as the second argument to ψ_B, we would not be able to build a parametricity witness relating a and this value, since a is only a local variable about which we have no other information. We therefore need to enrich the parametericity witness type further, for example by linking the relation and the function. The witness — or rather one of the oriented witnesses — becomes a dependent pair with three values:

$$\Sigma(R : A \to B \to \square)(\phi : A \to B).\, \Pi(a : A).\, R\,a\,(\phi\,a)$$

This new witness then provides a means of constructing the missing witness to instantiate ψ_B above and creating the function on the dependent products. However, as the parametricity witness has been enriched, we also need to prove the last property on the dependent product, which also requires more information. The univalent parametricity witness can be seen as the culmination of an iteration on this problem. We obtain a witness that is both symmetrical and stable through translation, *i. e.*, that passes through all the constructions without changing its nature.

8.2.2 Type equivalence and univalence

At the heart of univalent parametricity lies the principle of univalence, defined using type equivalence, a widespread notion with many existing definitions. Here, we explain the definitions chosen by Tabareau *et al.*, that form a basis for our work: isomorphism, equivalence, univalence. These are classic definitions that can be found in the Homotopy Type Theory book [64].

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

Definition 8.2.1 (Isomorphism)
A function $\phi : A \to B$ is an *isomorphism*, denoted $\mathsf{IsIso}(\phi)$, if there exists another function ψ that is both a left-inverse and a right-inverse for ϕ:

$$\mathsf{IsIso}(\phi) \quad := \quad \Sigma(\psi : B \to A).\ (\psi \circ \phi \doteqdot \mathrm{id}) \times (\phi \circ \psi \doteqdot \mathrm{id})$$

Note that type embeddings in TRAKT, introduced in § 6.1.1, are defined using an isomorphism in the case of total embeddings.

DEFINITION 8.2.2 (Equivalence)
The addition of an *adjunction* property to an isomorphism $\phi : A \to B$ gives an *equivalence*, denoted $\mathrm{IsEquiv}(\phi)$:

$$\begin{aligned}\mathrm{IsEquiv}(\phi) \quad := \quad & \Sigma(\psi : B \to A) \\ & (\mathrm{sec} : \psi \circ \phi \doteqdot \mathrm{id}) \\ & (\mathrm{ret} : \phi \circ \psi \doteqdot \mathrm{id}). \\ & \quad \mathrm{ap}\,\phi \circ \mathrm{sec} \doteqdot \mathrm{ret} \circ \phi\end{aligned}$$

where $\mathrm{ap}\, f\, p : f\, x = f\, y$ for $p : x = y$.

The adjunction property ties together the proofs of *section* and *retraction* by showing that a proof of $\phi \circ \psi \circ \phi \doteqdot \phi$ can be obtained by removing in the left-hand member either the outer composition $\phi \circ \psi$ using the retraction property, or the inner composition $\psi \circ \phi$ using the section property.

Sometimes, it is possible to define the inverse function as well as the section and retraction proofs, yet adjunction is hard to prove. In this case, there is a classic method in HoTT to obtain an equivalence from an isomorphism.

LEMMA 8.2.3
An isomorphism is an equivalence.

DEFINITION 8.2.4 (Type equivalence)
We say that two types A and B are equivalent, denoted $A \simeq B$, when there exists an equivalence $\phi : A \to B$:

$$A \simeq B \quad := \quad \Sigma\phi : A \to B.\, \mathrm{IsEquiv}(\phi)$$

A type equivalence $e : A \simeq B$ thus includes two *transport functions*, that we can also denote $\uparrow_e : A \to B$ and $\downarrow_e : B \to A$. They can be used to perform proof transfer from type A to type B, using $\uparrow_e$ at covariant occurrences, and $\downarrow_e$ at contravariant ones.[8]

8: This operation is similar to what happens in TRAKT, where the role of transport functions is played by embedding functions.

The *univalence principle* asserts that equivalent types are indistinguishable.

DEFINITION 8.2.5 (Univalence principle)
For any two types A and B, the canonical map of type $A = B \to A \simeq B$ is an equivalence.

In variants of CC_ω, the univalence principle can be postulated as an axiom, with no explicit computational content, as done for instance in the HoTT library [65] for the COQ proof assistant. Some more recent variants of dependent type theory feature a built-in computational univalence principle, and are used to implement experimental proof assistants, such as CUBICAL AGDA. In both cases, the univalence principle is a powerful proof transfer principle from $\square$ to $\square$, as for any two types A and B such that $A \simeq B$, and any $P : \square \to \square$, we can obtain that $P\, A \simeq P\, B$ as a direct corollary of univalence.[9] Concretely, $P\, B$ is obtained from $P\, A$ by appropriately allocating the transport functions provided by the equivalence proofs, a transfer process typically useful in the context of proof en-

[65]: BAUER *et al.* (2017), "The HoTT library: a formalization of homotopy type theory in Coq"

9: For $e : A \simeq B$ and u a proof of the univalence principle applied to A and B, we have:

$$\uparrow_u (\mathrm{ap}\, P\, (\downarrow_u e)) : P\, A \simeq P\, B.$$

gineering [66].

[66]: RINGER *et al.* (2021), "Proof repair across type equivalences"

8.2.3 Univalent parametricity translation

The key observation of univalent parametricity is that it is possible to enrich parametricity witnesses while preserving the abstraction theorem (8.1.1). Indeed, a raw parametricity witness links two types by an arbitrary relation, whereas univalent parametricity requires that the relation be an equivalence between these types. However, this enrichment requires a careful design in the translation of universes.

DEFINITION 8.2.6
The relational interpretation of the universe in univalent parametricity is the following:

$$\begin{aligned}\mathrm{Param}_i\ (A\,B : \square_i) \quad &:= \quad \Sigma(R : A \to B \to \square_i)(e : A \simeq B).\\ &\qquad \Pi(a : A)(b : B).\, R\,a\,b \simeq (a = {\downarrow_e}\ b)\end{aligned}$$

This type packages a relation R and an equivalence e such that R is equivalent to the functional relation associated with $\downarrow_e$. A crucial property of this new translation is the following:

LEMMA 8.2.7
Under the univalence axiom, the interpretation of the universe in univalent parametricity is equivalent to equivalence, *i. e.*, there exists a term $\mathrm{ParamEquiv}_i$ such that

$$\mathrm{ParamEquiv}_i : \Pi(A\,B : \square_i).\, \mathrm{Param}_i\ A\,B \simeq (A \simeq B).$$

Proof. Let A and B be two types. We have:

$$\begin{aligned}
&\mathrm{Param}_i\ A\,B\\
&\quad \downarrow \text{definition}\\
\equiv\ \ &\Sigma(R : A \to B \to \square_i)(e : A \simeq B).\,\Pi(a : A)(b : B).\,R\,a\,b \simeq (a = {\downarrow_e}\ b)\\
&\quad \downarrow \text{swapping mutually non-dependent binders}\\
\simeq\ \ &\Sigma(e : A \simeq B)(R : A \to B \to \square_i).\,\Pi(a : A)(b : B).\,R\,a\,b \simeq (a = {\downarrow_e}\ b)\\
&\quad \downarrow \text{univalence principle}\\
\simeq\ \ &\Sigma(e : A \simeq B)(R : A \to B \to \square_i).\,\Pi(a : A)(b : B).\,R\,a\,b = (a = {\downarrow_e}\ b)\\
&\quad \downarrow \text{equality in point-free style}\\
\simeq\ \ &\Sigma(e : A \simeq B)(R : A \to B \to \square_i).\,R = \lambda(a : A)(b : B).\,(a = {\downarrow_e}\ b)\\
&\quad \downarrow \text{contractible type in the right-hand member of the dependent pair}\\
\simeq\ \ &A \simeq B
\end{aligned}$$

■

This observation is actually an instance of a more general technique available for constructing syntactic models of type theory [59]. In fact, enriching the parametricity witness on the universe changes the structure of all the parametricity witnesses on types, making them dependent pairs, unlike the raw translation where they are relations. In this state, the translation is ill-formed and the abstraction theorem becomes invalid. In these models, a standard way to recover

[59]: BOULIER *et al.* (2017), "The next 700 syntactical models of type theory"

the abstraction theorem then consists in refining the translation into two variants, in order to handle correctly terms that are also types. Thus, the translation of $T : \square_i$ as a *term*, denoted $[T]_u$, is indeed a dependent pair, that contains a relation as well as the additional data prescribed by the interpretation of the universe Param_i. The translation of T as a *type*, $[\![T]\!]_u$, will be the relation itself, *i. e.*, the projection of the dependent pair $[T]_u$ onto its first component. The full univalent parametricity translation is therefore the following:

$$\begin{aligned}
[\![\langle\rangle]\!]_u &:= \langle\rangle \\
[\![\Gamma, x : A]\!]_u &:= [\![\Gamma]\!]_u, x : A, x' : A', x_R : [\![A]\!]_u\, x\, x' \\
[\![A]\!]_u &:= [A]_u.1 \\
[\square_i]_u &:= (\mathrm{Param}_i\, ;\, \mathrm{Equiv}_{\square_i}\, ;\, \mathrm{Coh}_{\square_i}) \\
[x]_u &:= x_R \\
[\Pi x : A.\, B]_u &:= \begin{pmatrix} R_\Pi\, A\, B\, ; \\ \mathrm{Equiv}_\Pi\, A\, A'\, [\![A]\!]_u\, B\, B'\, [\![B]\!]_u\, ; \\ \mathrm{Coh}_\Pi\, A\, A'\, [\![A]\!]_u\, B\, B'\, [\![B]\!]_u \end{pmatrix} \\
[\lambda x : A.\, t]_u &:= \lambda(x : A)(x' : A')(x_R : [\![A]\!]_u\, x\, x').\, [t]_u \\
[f\, t]_u &:= [f]_u\, t\, t'\, [t]_u
\end{aligned}$$

Figure 8.2: Univalent parametricity translation for CC_ω.

The most interesting cases are the universe and the dependent product, the other cases being similar to the raw translation. Being types, their translation is therefore a dependent triplet, the first component being a relation, the second a proof of equivalence, and the last a proof of coherence between the two preceding terms. The R_Π relation has the same structure as in the raw translation, using the univalent translation for the domain and the codomain:

$$R_\Pi\, A\, B \quad := \quad \lambda f\, f'.\, \Pi(x : A)(x' : A')(x_R : [\![A]\!]_u\, x\, x').\, [\![B]\!]_u\, (f\, x)\, (f'\, x')$$

Equivalence proofs — $\mathrm{Equiv}_{\square_i}$ and Equiv_Π — and coherence proofs — $\mathrm{Coh}_{\square_i}$ and Coh_Π — are available in the article [14].

[14]: TABAREAU *et al.* (2021), "The marriage of univalence and parametricity"

We can now phrase the abstraction theorem for univalent parametricity, where $\vdash_u$ refers to a typing judgment assuming the univalence axiom:

THEOREM 8.2.8 (Abstraction theorem for univalent parametricity)
If $\Gamma \vdash t : T$, then $[\![\Gamma]\!]_u \vdash_u [t]_u : [\![T]\!]_u\, t\, t'$.

We still note that in order to respect the abstraction theorem, the definition of $[\square_i]_u$ uses the univalence principle in an essential way. Indeed, since the relation on the universe is Param_i, we must have:

$$\begin{aligned}
& [\square_i]_u : [\![\square_{i+1}]\!]_u\, \square_i\, \square_i \\
\textit{i. e.}\quad & [\square_i]_u : \mathrm{Param}_{i+1}\, \square_i\, \square_i
\end{aligned}$$

The equivalence between a universe and itself, $\mathrm{Equiv}_{\square_i}$, is trivial and uses identity as both transport functions. Thus, proving the coherence property $\mathrm{Coh}_{\square_i}$ amounts to proving that the relation is equivalent to equality over the universe,

i. e.:

$$\Pi A\, B : \Box_i.\, \mathrm{Param}_i\, A\, B \simeq (A = B).$$

The proof is based on Lemma 8.2.7 and definitely requires the univalence axiom.

8.2.4 Omnipresence of the univalence axiom

Let us take the following example of goal:

$$\Pi(P : \mathbb{N} \to \Box).\, P\, 0 \to P\, 0$$

If we associate $\mathbb{N}$ with another type, for example a binary encoding N of natural numbers, then the goal associated by parametricity will be as follows, where 0_N is the constant associated with 0:

$$\Pi(P' : N \to \Box).\, P'\, 0_N \to P'\, 0_N$$

The univalent parametricity witness is built by induction on the syntax of the initial goal. During the traversal of this term, we are forced to translate $\Box$, thus invoking the proof $\mathrm{Coh}_\Box$ that requires the univalence axiom. However, we are in a case in which the proof of implication between the two goals is feasible without an axiom. The proof to be performed is as follows:

$$\frac{H' : \Pi(P' : N \to \Box).\, P'\, 0_N \to P'\, 0_N \qquad P : \mathbb{N} \to \Box \qquad p_0 : P\, 0}{P\, 0}$$

The trivial proof p_0 is possible here, but the general proof valid for other goals is the one using H', by placing embedding functions in one direction or the other according to the types to be inhabited. We then instantiate H' with:

$$P' := P \circ \downarrow_{\mathbb{N}}$$

If we define embedding functions that send the zero of one type to the zero of the other, then the second argument of H' can be p_0 unchanged. In this way, we have a proof of implication between the two goals without using an axiom. This case occurs as soon as an instance of $\Box$ is present in the initial goal without being problematic for the development of a manual proof.

Type equivalence in kit | 9

As explained in the previous chapter, parametricity provides a general framework to link terms in a λ-calculus, the most advanced example being univalent parametricity. This very powerful translation makes it possible, at the cost of adding an axiom, to generate proofs of equivalence by induction on the syntax, from any term in CC_ω. By adding constants to the calculus, it is possible to implement a tool in which the user can add heterogeneous parametricity witnesses, *i. e.*, equivalences between different types, before the start of the translation, allowing the generation of equivalences between different goals and thus proof transfer.

However, the need to add the univalence axiom to the calculus is an issue for two reasons. Firstly, as shown in § 8.2.4, many goals could be preprocessed with the same result as a univalent parametricity translation, but without using the univalence axiom, whereas univalent parametricity makes systematic use of it. Secondly, with the pragmatic aim of implementing a new parametricity plugin to ease proof transfer in Coq, it is necessary to ensure that the design space of the parametricity translation is logically consistent with the underlying logical theory of the proof assistant. Yet, the univalence axiom might introduce incompatibilities with the standard version of Coq.[1]

This chapter therefore presents a new parametricity relation, based on a new formulation of type equivalence (§ 9.1) that exposes all the information in a symmetrical and atomic way, as opposed to the classic formulation presented in Definition 8.2.4. The particularity of parametricity witnesses in this framework is that they contain a variable amount of information, ranging from the raw parametricity witness in the weakest case to the univalent witness in the strongest case. As a result, there is not a single stratified parametricity translation, but a set of possible associations (§ 9.2), the aim being to modulate the size of the parametricity witness and avoid depending on the univalence axiom when it is possible.

9.1 A new formulation of type equivalence . . . 82
9.1.1 Decomposing equivalence . . 83
9.1.2 Hierarchical recomposition of parametricity witnesses 86
9.2 Populating the hierarchy of relations . . . 87
9.2.1 Translation of universes 87
9.2.2 Translation of dependent products . . . 88
9.2.3 The case of non-dependent products . . . 89

1: One generally uses it in the HoTT library [65] in order to work in a fully controlled context.

9.1 A new formulation of type equivalence

As shown previously, Definition 8.2.6 describes a univalent parametricity witness both very rich — it systematically requires equivalence — and very dense — it contains only three values. As a result, the coherence condition in the case of the universe requires the definition of the witness to be equivalent to equality between types, which forces the translation to resort to the univalence axiom systematically. However, some goals contain occurrences of $\square$ for which it is excessive to require equivalence in order to perform preprocessing.

The situation suggests to search for a hybrid parametricity relation, needing equivalence only in cases where it is strictly necessary, and requiring less information where possible. This involves a decomposition of type equivalence (§ 9.1.1), *i. e.*, spreading out the information it contains. Once the decomposition is done, it is possible to carve a hierarchy of parametricity witnesses by selectively picking values from this Σ-type (§ 9.1.2).

9.1.1 Decomposing equivalence

Let us first observe that the Definition 8.2.4 of type equivalence is quite asymmetrical, although this fact is somehow put under the rug by the infix $A \simeq B$ notation. Indeed, first, the data of an equivalence $e : A \simeq B$ privilege the left-to-right direction, as $\uparrow_e$ is directly accessible from e as its first projection, while accessing the right-to-left transport requires an additional projection. Second, the statement of the adjunction property, available in Definition 8.2.2, is:

$$\mathrm{ap}\,\phi \circ \mathrm{sec} \doteqdot \mathrm{ret} \circ \phi$$

This statement uses proofs sec and ret, respectively the section and retraction properties of e, but not in a symmetrical way, although swapping them provides an equivalent definition. This entanglement prevents any hope to trace the respective roles of each transport function during the course of a given univalent parametricity translation. Exercise 4.2 in the HoTT book [64] however suggests a symmetrical wording of the definition of type equivalence, in terms of functional relations.

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

Definition 9.1.1
Any relation $R : A \to B \to \square_i$ is *functional* when each value of A is uniquely linked to exactly one value of B in R:

$$\mathsf{IsFun}(R) \quad := \quad \Pi a : A.\, \mathsf{IsContr}(\Sigma b : B.\, R\, a\, b)$$

where $\mathsf{IsContr}(\cdot)$ is the standard contractibility predicate:

$$\mathsf{IsContr}(T) \quad := \quad \Sigma t_0 : T.\, \Pi t : T.\, t = t_0$$

We can now obtain an equivalent but symmetrical characterisation of type equivalence, as a functional relation whose symmetrisation is also functional.

Lemma 9.1.2
For any types $A, B : \square_i$, the type $A \simeq B$ is equivalent to:

$$\Sigma R : A \to B \to \square_i.\, \mathsf{IsFun}(R) \times \mathsf{IsFun}(R^{-1})$$

where relation $R^{-1} : B \to A \to \square_i$ just swaps the arguments of an arbitrary $R : A \to B \to \square_i$.

Let us sketch a proof of this result, left as an exercise in [64].

We need the following lemma, that explains why $\mathsf{IsFun}(\cdot)$ characterises functional relations:

Lemma 9.1.3
For any types $A, B : \square_i$, we have:

$$(A \to B) \quad \simeq \quad \Sigma R : A \to B \to \square_i.\, \mathsf{IsFun}(R).$$

Proof. The proof goes by chaining the following equivalences:

$$
\begin{array}{ll}
& \Sigma R : A \to B \to \square_i.\, \mathsf{IsFun}(R) \\
& \quad \downarrow \text{definition} \\
\equiv & \Sigma R : A \to B \to \square_i.\, \Pi a : A.\, \mathsf{IsContr}(\Sigma b : B.\, R\, a\, b) \\
& \quad \downarrow \text{swapping mutually non-dependent binders} \\
\simeq & \Pi a : A.\, \Sigma R : A \to B \to \square_i.\, \mathsf{IsContr}(\Sigma b : B.\, R\, a\, b) \\
& \quad \downarrow \text{by defining } P := R\, a\text{, the first binder is no longer dependent} \\
\simeq & A \to \Sigma P : B \to \square_i.\, \mathsf{IsContr}(\Sigma b : B.\, P\, b) \\
& \quad \downarrow \text{standard HoTT lemma} \\
\simeq & A \to B
\end{array}
$$

■

Proof of Lemma 9.1.2. The proof is done by chaining the following equivalences:

$$
\begin{array}{ll}
& A \simeq B \\
& \quad \downarrow \text{definition} \\
\simeq & \Sigma f : A \to B.\, \mathsf{IsEquiv}(f) \\
& \quad \downarrow \text{classic result of HoTT} \\
\simeq & \Sigma f : A \to B.\, \Pi b : B.\, \mathsf{IsContr}(\Sigma a : A.\, f\, a = b) \\
& \quad \downarrow \text{definition of } \mathsf{IsFun}(\cdot) \\
\simeq & \Sigma f : A \to B.\, \mathsf{IsFun}(\lambda (b : B)(a : A).\, f\, a = b) \\
& \quad \downarrow \text{Lemma 9.1.3} \\
\simeq & \Sigma (f : \Sigma (R : A \to B \to \square_i).\, \mathsf{IsFun}(R)).\, \mathsf{IsFun}(f.1^{-1}) \\
& \quad \downarrow \text{associativity of } \Sigma \\
\simeq & \Sigma R : A \to B \to \square_i.\, \mathsf{IsFun}(R) \times \mathsf{IsFun}(R^{-1})
\end{array}
$$

■

The symmetrical version of type equivalence provided by Lemma 9.1.2 however does not expose explicitly the two transport functions in its data, although this computational content can be extracted *via* projections on contractibility proofs. In fact, it is possible to devise a definition of type equivalence that directly provides the two transport functions in its data, while remaining symmetrical. The essential ingredient of this rewording is the alternative characterisation of functional relations.

Definition 9.1.4
For any types $A, B : \square_i$, a relation $R : A \to B \to \square_i$ is a *univalent map*, denoted $\mathsf{IsUMap}(R)$, when there exists a function m whose graph is exactly described by R, and this very property comes with a coherence condition:

$$
\begin{array}{lll}
\mathsf{IsUMap}(R) & := & \Sigma (m : A \to B) \\
& & (\pi_1 : \Pi (a : A)(b : B).\, m\, a = b \to R\, a\, b) \\
& & (\pi_2 : \Pi (a : A)(b : B).\, R\, a\, b \to m\, a = b). \\
& & \quad \Pi (a : A)(b : B).\, (\pi_1\, a\, b) \circ (\pi_2\, a\, b) \doteqdot id
\end{array}
$$

Now comes the crux lemma of this subsection, formally proved in the code of Trocq:

Lemma 9.1.5
For any types $A, B : \Box_i$ and any relation $R : A \to B \to \Box_i$,

$$\mathsf{IsFun}(R) \simeq \mathsf{IsUMap}(R).$$

Proof. The proof goes by rewording the left hand side step by step:

$$\begin{aligned}
& \mathsf{IsFun}(R) \\
& \quad \downarrow \text{definitions} \\
\simeq\ & \Pi a_0.\, \Sigma(c : \Sigma b.\, R\, a_0\, b).\, \Pi(p : \Sigma b.\, R\, a_0\, b).\, c = p \\
& \quad \downarrow \text{associativity of } \Sigma \\
\simeq\ & \Pi a_0.\, \Sigma b_0.\, \Sigma r : R\, a_0\, b_0.\, \Pi(p : \Sigma b.\, R\, a_0\, b).\, (b_0\, ;\, r) = p \\
& \quad \downarrow \text{intuitionistic choice} \\
\simeq\ & \Sigma \phi : A \to B.\, \Pi a_0.\, \Sigma r : R\, a_0\, (\phi\, a_0).\, \Pi(p : \Sigma b.\, R\, a_0\, b).\, (\phi\, a_0\, ;\, r) = p \\
& \quad \downarrow \text{swapping of binders and abstraction of } a_0 \text{ in } r \\
\simeq\ & \Sigma \phi.\, \Sigma(r : \Pi a.\, R\, a\, (\phi\, a)).\, \Pi a_0.\, \Pi(p : \Sigma b.\, R\, a_0\, b).\, (\phi\, a_0\, ;\, r\, a_0) = p \\
& \quad \downarrow \text{splitting } p \\
\simeq\ & \Sigma \phi.\, \Sigma r.\, \Pi a_0.\, \Pi b_0.\, \Pi(r' : R\, a_0\, b_0).\, (\phi\, a_0\, ;\, r\, a_0) = (b\, ;\, r') \\
& \quad \downarrow \text{decomposition of the equality proof over a dependent pair} \\
\simeq\ & \Sigma \phi.\, \Sigma r.\, \Pi a_0.\, \Pi b_0.\, \Pi r'.\, \Sigma e : \phi\, a_0 = b_0.\, r\, a_0 =_e r' \\
& \quad \downarrow \text{swapping of binders and abstraction of } a_0,\ b_0, \text{ and } r' \text{ in } e \\
\simeq\ & \Sigma \phi.\, \Sigma r.\, \Sigma(e : \Pi a.\, \Pi b.\, R\, a\, b \to \phi\, a = b).\, \Pi a_0.\, \Pi b_0.\, \Pi r'.\, r =_{e\, a_0\, b_0\, r'} r'
\end{aligned}$$

We identify that ϕ is value m in the definition of $\mathsf{IsUMap}(R)$, and e is value π_2. By renaming values and reorganising Σ-types, we are left to show the following property:

$$\begin{aligned}
& \Sigma(\pi_1 : \Pi a.\, \Pi b.\, m\, a = b \to R\, a\, b).\, (\pi_1\, a_0\, b_0) \circ (\pi_2\, a_0\, b_0) \doteqdot \mathrm{id} \\
\simeq\ & \Sigma(r : \Pi a.\, R\, a\, (m\, a)).\, \Pi(r' : R\, a_0\, b_0).\, r\, a_0 =_{\pi_2\, a_0\, b_0\, r'} r'
\end{aligned}$$

We refer the reader to the companion code. ■

As a direct corollary, we obtain a novel characterisation of type equivalence:

Theorem 9.1.6
For any types $A, B : \Box$, we have:

$$(A \simeq B) \simeq \mathrm{Param}^{\top}\, A\, B$$

where relation $\mathrm{Param}^{\top}\, A\, B$ is defined as:

$$\begin{aligned}
\mathrm{Param}^{\top}\, A\, B \quad := \quad & \Sigma R : A \to B \to \Box. \\
& \mathsf{IsUMap}(R) \times \mathsf{IsUMap}(R^{-1})
\end{aligned}$$

The resulting collection of data is now symmetrical, as the reverse direction of the equivalence based on univalent maps can be obtained by flipping the relation and swapping the two functionality proofs. If the η rule for records is verified, symmetry is even definitionally involutive.

9.1.2 Hierarchical recomposition of parametricity witnesses

Definition 9.1.4 of univalent maps and the resulting rephrasing of type equivalence suggest introducing a hierarchy of structures for heterogeneous relations, that explains how close a given relation is to type equivalence. In turn, this distance is described in terms of structure available respectively on the left-to-right and right-to-left transport functions.

DEFINITION 9.1.7
For $n, k \in \{0, 1, 2_a, 2_b, 3, 4\}$, and $\alpha = (n, k)$, relation Param^α is:

$$\begin{aligned}\mathrm{Param}^\alpha \quad &:= \quad \lambda(A\,B : \square).\\ &\qquad \Sigma R : A \to B \to \square.\, \mathrm{Class}_\alpha\, R\end{aligned}$$

where the *map class* $\mathrm{Class}_\alpha\, R$ itself unfolds to a pair type of two *unilateral witnesses* — one from A to B, one from B to A:

$$(\mathsf{M}_n\, A\, B\, R) \times (\mathsf{M}_k\, B\, A\, R^{-1})$$

with M_i defined as:

$$\begin{aligned}
\mathsf{M}_0\, A\, B\, R &:= \quad .\\
\mathsf{M}_1\, A\, B\, R &:= \quad A \to B\\
\mathsf{M}_{2_a}\, A\, B\, R &:= \quad \Sigma m : A \to B.\, G_{2_a}\, A\, B\, m\, R\\
\mathsf{M}_{2_b}\, A\, B\, R &:= \quad \Sigma m : A \to B.\, G_{2_b}\, A\, B\, m\, R\\
\mathsf{M}_3\, A\, B\, R &:= \quad \Sigma(m : A \to B).\\
&\qquad (G_{2_a}\, A\, B\, m\, R) \times (G_{2_b}\, A\, B\, m\, R)\\
\mathsf{M}_4\, A\, B\, R &:= \quad \Sigma(m : A \to B)\\
&\qquad (g_a : G_{2_a}\, A\, B\, m\, R)\\
&\qquad (g_b : G_{2_b}\, A\, B\, m\, R).\\
&\qquad\quad \Pi a\, b.\, (g_a\, a\, b) \circ (g_b\, a\, b) \doteqdot id
\end{aligned}$$

with

$$\begin{aligned}
G_{2_a}\, A\, B\, m\, R &:= \quad \Pi(a : A)(b : B).\, m\, a = b \to R\, a\, b\\
G_{2_b}\, A\, B\, m\, R &:= \quad \Pi(a : A)(b : B).\, R\, a\, b \to m\, a = b
\end{aligned}$$

For any types A and B, and any $r : \mathrm{Param}^\alpha\, A\, B$, we will use notations $\mathsf{rel}(r)$, $\mathsf{map}(r)$ and $\mathsf{comap}(r)$ to refer respectively to the relation, function of type $A \to B$, function of type $B \to A$, included in the data of r, for a suitable α.

DEFINITION 9.1.8
We denote $\mathcal{A}$ the set $\{0, 1, 2_a, 2_b, 3, 4\}^2$, used to index map classes in Definition 9.1.7. This set is partially ordered for the product order on $\{0, 1, 2_a, 2_b, 3, 4\}$ defined from the partial order $0 < 1 < 2_* < 3 < 4$ for 2_* either 2_a or 2_b, and with 2_a and 2_b being incomparable.

REMARK 9.1.9
Relation $\mathrm{Param}^{(4,4)}$ of Definition 9.1.7 coincides with the relation $\mathrm{Param}^\top$ introduced in Theorem 9.1.6, equivalent to the univalent parametricity witness type. Similarly, we denote as $\mathrm{Param}^\perp$ the relation $\mathrm{Param}^{(0,0)}$, that amounts to just

having a relation $R : A \to B \to \square$ as in the raw parametricity translation. A relation equipped with structure $\mathrm{Param}^{(4,0)}\ A\ B$ (respectively $\mathrm{Param}^{(3,3)}\ A\ B$) is the graph of a univalent map from A to B (respectively an isomorphism between A and B).

In the associated code, the corresponding lattice to the collection of M_n is implemented as a hierarchy of dependent tuples — more precisely, of record types. Each arrow of Figure 9.1 represents an inclusion of the data packed in the source structure into the data packed in the target one. Moreover, nodes are labeled with the names of the corresponding record fields introduced by the richer structure.

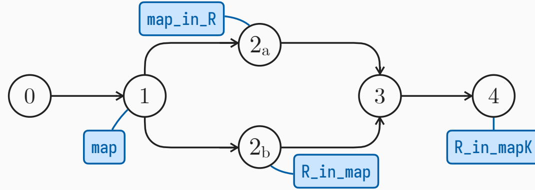


Figure 9.1: Implementation of the hierarchy of Definition 9.1.7

9.2 Populating the hierarchy of relations

We shall now revisit the parametricity translations of § 8. In particular, combining Theorem 9.1.6 with the abstraction theorem for univalent parametricity ensures the existence of a term $p_{\square_i}$ such that:

$$\vdash_u p_{\square_i} : \mathrm{Param}^{\top}_{i+1}\, \square_i\, \square_i \quad \text{and} \quad \mathrm{rel}(p_{\square_i}) \simeq \mathrm{Param}^{\top}_i.$$

Otherwise said, relation $\mathrm{Param}^{\top} : \square \to \square \to \square$ can be endowed with a $\mathrm{Param}^{\top}$ structure, assuming univalence. Similarly, the equation for universes, in Figure 8.1 describing the raw parametricity translation, can be read as the fact that relation $\mathrm{Param}^{\bot}$ on universes can be endowed with a $\mathrm{Param}^{\bot}\, \square\, \square$ structure.

9.2.1 Translation of universes

Now the hierarchy of structures introduced by Definition 9.1.7 enables a finer grained analysis of the possible relational interpretations of universes. Not only would this put the raw and univalent parametricity translations under the same hood, but it would also allow for generalising parametricity to a larger class of relations. For this purpose, we generalise the previous observation, on the key ingredient for translating universes: for each $\alpha \in \mathcal{A}$, relation Param^{α} may be endowed with several structures from the lattice, and we need to study which ones, depending on α. Otherwise said, we need to identify the pairs $(\alpha, \beta) \in \mathcal{A}^2$ for which it is possible to construct a term $p_{\square}^{\alpha,\beta}$ such that:

$$\vdash_u p_{\square}^{\alpha,\beta} : \mathrm{Param}^{\beta}\, \square\, \square \quad \text{and} \quad \mathrm{rel}(p_{\square}^{\alpha,\beta}) \equiv \mathrm{Param}^{\alpha} \tag{9.1}$$

Note that here we aim at a definitional equality between $\mathrm{rel}(p_{\square}^{\alpha,\beta})$ and Param^{α}, rather than an equivalence. It is easy to see that a term $p_{\square}^{\alpha,\bot}$ exists for any $\alpha \in \mathcal{A}$,

as $\mathrm{Param}^{\perp}$ requires no structure on the relation. On the other hand, it is not possible to construct a term $p_{\square}^{\perp,\top}$, *i. e.*, to turn an arbitrary relation into a type equivalence.

Definition 9.2.1
We denote as $\mathcal{D}_{\square}$ the following subset of $\mathcal{A}^2$:

$$\mathcal{D}_{\square} = \{(\alpha,\beta) \in \mathcal{A}^2 \mid \alpha = \top \vee \beta \in \{0,1,2_{\mathrm{a}}\}^2\}$$

The associated code [2] constructs terms $p_{\square}^{\alpha,\beta}$ for every pair $(\alpha,\beta) \in \mathcal{D}_{\square}$, using a meta-program to generate them from a minimal collection of manual definitions. In particular, assuming univalence, it is possible to construct a term $p_{\square}^{\top,\top}$, that can be seen as an analogue of the translation $[\square]$ of univalent parametricity. More generally, the provided terms $p_{\square}^{\alpha,\beta}$ depend on univalence if and only if $\beta \notin \{0,1,2_{\mathrm{a}}\}^2$.

2: File Param_Type.v.

9.2.2 Translation of dependent products

The next natural question is the study of the possible structures Param^{γ} that can equip a relation associated with a product type $\Pi x : A.\, B$, when the relations associated with types A and B are respectively equipped with structures Param^{α} and Param^{β}.

Otherwise said, we need to identify the triples $(\alpha,\beta,\gamma) \in \mathcal{A}^3$ for which it is possible to construct a term p_{Π}^{γ} such that:

$$\frac{\Gamma \vdash A_R : \mathrm{Param}^{\alpha}\, A\, A' \qquad \Gamma, x : A, x' : A', x_R : A_R\, x\, x' \vdash B_R : \mathrm{Param}^{\beta}\, B\, B'}{\Gamma \vdash p_{\Pi}^{\gamma}\, A_R\, B_R : \mathrm{Param}^{\gamma}\, (\Pi x : A.\, B)\, (\Pi x' : A'.\, B')} \quad \text{and}$$

$$\mathsf{rel}(p_{\Pi}^{\gamma}\, A_R\, B_R) \quad \equiv \quad \lambda f\, f'.\, \Pi(x : A)(x' : A')(x_R : \mathsf{rel}(A_R)\, x\, x').\ \mathsf{rel}(B_R)\,(f\, x)\,(f\, x')$$

The corresponding collection of triples can actually be described as a function $\mathcal{D}_{\Pi} : \mathcal{A} \to \mathcal{A}^2$, such that $\mathcal{D}_{\Pi}(\gamma) = (\alpha,\beta)$ provides the *minimal* requirements on the structures associated with A and B, with respect to the partial order on $\mathcal{A}^2$. The associated code [3] provides a corresponding collection of terms p_{Π}^{γ} for each $\gamma \in \mathcal{A}$, as well as all the associated weakenings. Once again, these definitions are generated by a meta-program. Observe in particular that by symmetry, $p_{\Pi}^{(m,n)}$ can be obtained from $p_{\Pi}^{(m,0)}$ and $p_{\Pi}^{(n,0)}$ by swapping the latter and glueing it to the former. Therefore, the values of p_{Π}^{γ} and $\mathcal{D}_{\Pi}(\gamma)$ are completely determined by those of $p_{\Pi}^{(m,0)}$ and $\mathcal{D}_{\Pi}(m,0)$. In particular, for any $m,n \in \mathcal{A}$:

3: File Param_forall.v.

$$\mathcal{D}_{\Pi}(m,n) = ((m_A,n_A),(m_B,n_B))$$

for $m_A, n_A, m_B, n_B \in \mathcal{A}$ defined as

$$\mathcal{D}_{\Pi}(m,0) = ((0,n_A),(m_B,0))$$
$$\mathcal{D}_{\Pi}(n,0) = ((0,m_A),(n_B,0))$$

We sum up in Figure 9.2 the values of $\mathcal{D}_{\Pi}(m,0)$.[4]

4: The greyed-out cells highlight a weaker dependency in the case of an arrow type compared with the general case of the dependent product.

m	$\mathcal{D}_\Pi(m,0)_1$	$\mathcal{D}_\Pi(m,0)_2$
0	$(0,0)$	$(0,0)$
1	$(0,2_a)$	$(1,0)$
2_a	$(0,4)$	$(2_a,0)$
2_b	$(0,2_a)$	$(2_b,0)$
3	$(0,4)$	$(3,0)$
4	$(0,4)$	$(4,0)$

m	$\mathcal{D}_\to(m,0)_1$	$\mathcal{D}_\to(m,0)_2$
0	$(0,0)$	$(0,0)$
1	$(0,1)$	$(1,0)$
2_a	$(0,2_b)$	$(2_a,0)$
2_b	$(0,2_a)$	$(2_b,0)$
3	$(0,3)$	$(3,0)$
4	$(0,4)$	$(4,0)$

Figure 9.2: Minimal dependencies for dependent and non-dependent products at class $(m,0)$

9.2.3 The case of non-dependent products

Note that in the case of a non-dependent product, constructing $p^\gamma_\to$ requires less structure on the domain A of an arrow type $A \to B$, which motivates the introduction of function $\mathcal{D}_\to(\gamma)$. Using the combinator for dependent products to interpret an arrow type, albeit correct, potentially pulls in unnecessary structure — and axiom — requirements. The associated code [5] includes a construction of terms $p^\gamma_\to$ for any $\gamma \in \mathcal{A}$.

5: File `Param_arrow.v`.

A calculus for proof transfer 10

10.1 Raw parametricity sequents . 90
10.2 Univalent parametricity sequents 92
10.3 Annotated type theory 93
10.4 The Trocq calculus 94
10.5 Constants 96

This chapter introduces Trocq, a framework for proof transfer designed as a generalisation of parametricity translations, so as to allow for interpreting types as instances of the structures introduced in § 9.2.1. We adopt a sequent style presentation, that closely fits the type system of CC_ω, while explaining in a consistent way the transformations of terms and contexts. This choice of presentation departs from the standard literature about parametricity in Pure Type Systems. Yet, it brings the presentation closer to actual implementations, whose necessary management of parametricity contexts is put under the rug by notational conventions.

For this purpose, we successively introduce four calculi, of increasing sophistication. We start with introducing this sequent style presentation by rephrasing the raw parametricity translation (§ 10.1), and the univalent parametricity one (§ 10.2). We then introduce CC_ω^+ , a calculus of constructions with annotations on sorts and subtyping (§ 10.3), before defining the Trocq calculus (§ 10.4).

10.1 Raw parametricity sequents

We introduce *parametricity contexts*, under the form of a list of triples packaging pairs of variables together with a witness that they are related:

$$\Xi ::= \varepsilon \mid \Xi,\ x\ \sim\ x' \because x_R$$

We write $(x, x', x_R) \in \Xi$ if there exists Ξ' and Ξ'' such that:

$$\Xi = \Xi',\ x\ \sim\ x' \because x_R,\ \Xi''$$

We denote $\mathrm{Var}(\Xi)$ the sequence of variables related in a parametricity context Ξ:

$$\mathrm{Var}(\varepsilon) = \varepsilon \qquad \mathrm{Var}(\Xi,\ x\ \sim\ x' \because x_R) = \mathrm{Var}(\Xi), x, x', x_R$$

A parametricity context Ξ is *well formed*, written $\Xi \vdash$, if the sequence $\mathrm{Var}(\Xi)$ is duplicate-free. In this case, we use the notation $\Xi(x) = (x', x_R)$ as a synonym of $(x, x', x_R) \in \Xi$.

A *parametricity judgment* relates a parametricity context Ξ and three terms M, M', M_R of CC_ω. Parametricity judgments are defined by rules of Figure 10.1. We denote and read them in the following way:

$$\Xi \vdash M\ \sim\ M' \because M_R$$

In context Ξ, term M translates to term M', because M_R .

Lemma 10.1.1
The relation associating a term M with pair (M', M_R) such that

$$\Xi \vdash M\ \sim\ M' \because M_R$$

$$\frac{}{\Xi \vdash \Box_i \sim \Box_i \because \lambda(A\,B : \Box_i).\, A \to B \to \Box_i} \text{(ParamSort)}$$

$$\frac{(x, x', x_R) \in \Xi \qquad \Xi \vdash}{\Xi \vdash x \sim x' \because x_R} \text{(ParamVar)}$$

$$\frac{\Xi \vdash M \sim M' \because M_R \quad \Xi \vdash N \sim N' \because N_R}{\Xi \vdash M\,N \sim M'\,N' \because M_R\,N\,N'\,N_R} \text{(ParamApp)}$$

$$\frac{\Xi, x \sim x' \because x_R \vdash M \sim M' \because M_R}{\Xi \vdash \lambda x : A.\, M \sim \lambda x' : A'.\, M' \because \lambda x\,x'\,x_R.\, M_R} \text{(ParamLam)}$$

$$\frac{\Xi \vdash A \sim A' \because A_R \qquad \Xi, x \sim x' \because x_R \vdash B \sim B' \because B_R \qquad x, x' \notin \mathrm{Var}(\Xi)}{\Xi \vdash \Pi x : A.\, B \sim \Pi x' : A'.\, B' \because \lambda f\,g.\, \Pi x\,x'\,x_R.\, B_R\,(f\,x)\,(g\,x')} \text{(ParamPi)}$$

Figure 10.1: Param: sequent-style binary parametricity translation

with Ξ a well-formed parametricity context, is *functional*: for any term M and any well-formed Ξ:

$$\begin{aligned} &\forall M', N', M_R, N_R, \\ &\quad \Xi \vdash M \sim M' \because M_R \wedge \Xi \vdash M \sim N' \because N_R \implies \\ &\qquad (M', M_R) = (N', N_R) \end{aligned}$$

Proof. Immediate by induction on the syntax of M. ■

This presentation of parametricity thus provides an alternative definition of translation $[\![\cdot]\!]$, from Figure 8.1, and accounts for the prime-based notational convention used in the latter.

Definition 10.1.2
A parametricity context Ξ is *admissible* for a well-formed typing context Γ, denoted $\Gamma \rhd \Xi$, when Ξ is well formed as a parametricity context and Γ provides consistent type annotations for all terms in Ξ, that is, for any variables x, x', x_R such that $\Xi(x) = (x', x_R)$, and for any terms A' and A_R:

$$\Xi \vdash \Gamma(x) \sim A' \because A_R \quad \implies \quad \Gamma(x') = A' \,\wedge\, \Gamma(x_R) \equiv A_R\,x\,x'$$

We can now state and prove an abstraction theorem:

Theorem 10.1.3 (Abstraction theorem)

$$\frac{\Gamma \vdash M : A \qquad \Gamma \rhd \Xi \qquad \Xi \vdash M \sim M' \because M_R \qquad \Xi \vdash A \sim A' \because A_R}{\Gamma \vdash M' : A' \quad \text{and} \quad \Gamma \vdash M_R : A_R\,M\,M'}$$

Proof. By induction on the derivation of $\Xi \vdash M \sim M' \because M_R$. ■

10.2 Univalent parametricity sequents

We now propose in Figure 10.2 a rephrased version of the univalent parametricity translation [14], using the same sequent style and replacing the translation of universes with the equivalent relation $\mathrm{Param}^{\top}$. In this variant, parametricity judgments are denoted in the following way, where Ξ is a parametricity context and M, M', and M_R are terms of CC_ω:

[14]: TABAREAU *et al.* (2021), "The marriage of univalence and parametricity"

$$\Xi \vdash_u M \sim M' \because M_R$$

The u index is a reminder that typing judgments $\Gamma \vdash_u M : A$ involved in the associated abstraction theorem are typing judgments of CC_ω augmented with the univalence axiom.

$$\frac{}{\Xi \vdash_u \square_i \sim \square_i \because p_{\square_i}^{\top,\top}}\ (\text{UParamSort})$$

$$\frac{(x, x', x_R) \in \Xi \qquad \Xi \vdash}{\Xi \vdash_u x \sim x' \because x_R}\ (\text{UParamVar})$$

$$\frac{\Xi \vdash_u M \sim M' \because M_R \qquad \Xi \vdash_u N \sim N' \because N_R}{\Xi \vdash_u M\,N \sim M'\,N' \because M_R\,N\,N'\,N_R}\ (\text{UParamApp})$$

$$\frac{\Xi \vdash_u A \sim A' \because A_R \qquad \Xi, x \sim x' \because x_R \vdash_u M \sim M' \because M_R}{\Xi \vdash_u \lambda x : A.\, M \sim \lambda x' : A'.\, M' \because \lambda x\, x'\, x_R.\, M_R}\ (\text{UParamLam})$$

$$\frac{\Xi \vdash_u A \sim A' \because A_R \qquad \Xi, x \sim x' \because x_R \vdash_u B \sim B' \because B_R}{\Xi \vdash_u \Pi x : A.\, B \sim \Pi x' : A'.\, B' \because p_\Pi^{\top}\, A_R\, B_R}\ (\text{UParamPi})$$

Figure 10.2: UParam: univalent parametricity rules

We can now rephrase the abstraction theorem for univalent parametricity.

Theorem 10.2.1 (Univalent abstraction theorem)

$$\frac{\Gamma \vdash M : A \qquad \Gamma \rhd \Xi \qquad \begin{matrix}\Gamma \vdash \\ \Xi \vdash_u M \sim M' \because M_R\end{matrix} \qquad \Xi \vdash_u A \sim A' \because A_R}{\Gamma \vdash M' : A' \qquad \text{and} \qquad \Xi \vdash_u M_R : \mathsf{rel}(A_R)\; M\; M'}$$

Proof. By induction on the derivation of $\Xi \vdash_u M \sim M' \because M_R$. ■

Remark 10.2.2

In Theorem 10.2.1, term $\mathsf{rel}(A_R)$ is a relation of type $A \to A' \to \square$. Indeed:

$$\frac{\Gamma \vdash A : \square_i \qquad \Xi \vdash_u A \sim A' \because A_R \qquad \Gamma \rhd \Xi}{\Gamma \vdash_u A_R : \mathsf{rel}(p_{\square_i}^{\top,\top})\; A\; A'}$$

entails A_R has type

$$\begin{aligned} & \mathsf{rel}(p_{\square_i}^{\top,\top})\; A\; A' \\ \equiv\;& \mathrm{Param}^{\top}\; A\; A' \\ \equiv\;& \Sigma R : A \to A' \to \square.\ \mathsf{IsUMap}(R) \times \mathsf{IsUMap}(R^{-1}) \end{aligned}$$

10.3 Annotated type theory

We are now ready to generalise the relational interpretation of types provided by the univalent parametricity translation, so as to allow for interpreting sorts with instances of weaker structures than equivalence. For this purpose, we introduce a variant CC_ω^+ of CC_ω where each universe is annotated with a label indicating the structure available on its relational interpretation. Recall from § 9.2.1 that we have used pairs $\alpha \in \mathcal{A}^2$ to identify the different structures of the lattice disassembling type equivalence: these are the labels annotating sorts of CC_ω^+, so that if A has type $\square^\alpha$, then the associated relation A_R has type $\mathrm{Param}^\alpha\, A\, A'$. The syntax of CC_ω^+ is thus:

$$M, N, A, B \in \mathcal{T}_{CC_\omega^+} ::= \square_i^\alpha \mid x \mid M\ N \mid \lambda x : A.\, M \mid \Pi x : A.\, B$$
$$\alpha \in \mathcal{A} = \{0, 1, 2_\mathrm{a}, 2_\mathrm{b}, 3, 4\}^2 \qquad i \in \mathbb{N}$$

Before completing the actual formal definition of the Trocq proof transfer framework, let us informally illustrate how these annotations shall drive the interpretation of terms, and in particular, of a dependent product $\Pi x : A.\, B$. In this case, before translating B, three terms representing the bound variable x, its translation x', and the parametricity witness x_R are added to the context. The type of x_R is $\mathsf{rel}(A_R)\, x\, x'$ where A_R is the parametricity witness relating A to its translation A'. The role of annotation α on the sort of type A is thus to govern the amount of information available in witness x_R, by determining the type of A_R. This intent is reflected in the typing rules of CC_ω^+, that rely on the definition of the loci $\mathcal{D}_\square$, $\mathcal{D}_\to$ and $\mathcal{D}_\Pi$, introduced in § 9.2.

Typing terms in CC_ω^+ requires defining a *subtyping* relation $\preccurlyeq$, defined by the rules of Figure 10.3. The typing rules of CC_ω^+ are available in Figure 10.4 and follow standard presentations [67]. The $\equiv$ relation in the SubConv rule is the *conversion* relation, defined as the closure of α-equivalence and β-reduction on this variant of λ-calculus. We hence have two types of judgment in this calculus:

[67]: Aspinall *et al.* (2001), "Subtyping dependent types"

$$\Gamma \vdash_+ A \preccurlyeq B \quad \text{and} \quad \Gamma \vdash_+ M : A$$

where M, A, and B are terms in CC_ω^+ and Γ is a context in CC_ω^+.[1]

1: $\Gamma ::= \varepsilon \mid \Gamma,\ x : A.$

$$\frac{\Gamma \vdash_+ A : K \qquad \Gamma \vdash_+ B : K \qquad A \equiv B}{\Gamma \vdash_+ A \preccurlyeq B}\ (\textsc{SubConv})$$

$$\frac{\alpha \geq \beta \qquad i \leq j}{\Gamma \vdash_+ \square_i^\alpha \preccurlyeq \square_j^\beta}\ (\textsc{SubSort}) \qquad \frac{\Gamma \vdash_+ M'\ N : K \qquad \Gamma \vdash_+ M \preccurlyeq M'}{\Gamma \vdash_+ M\ N \preccurlyeq M'\ N}\ (\textsc{SubApp})$$

$$\frac{\Gamma, x : A \vdash_+ M \preccurlyeq M'}{\Gamma \vdash_+ \lambda x : A.\, M \preccurlyeq \lambda x : A.\, M'}\ (\textsc{SubLam})$$

$$\frac{\Gamma \vdash_+ \Pi x : A.\, B : \square_i^\gamma \qquad \Gamma \vdash_+ A' \preccurlyeq A \qquad \Gamma, x : A' \vdash_+ B \preccurlyeq B'}{\Gamma \vdash_+ \Pi x : A.\, B \preccurlyeq \Pi x : A'.\, B'}\ (\textsc{SubPi})$$

$$K ::= \square_i^\gamma \mid \Pi x : A.\, K$$

Figure 10.3: Subtyping rules for CC_ω^+

$$\frac{\Gamma \vdash_+ M : A \qquad \Gamma \vdash_+ A \preccurlyeq B}{\Gamma \vdash_+ M : B}\ (\textsc{Conv}^+) \qquad \frac{(\alpha,\beta) \in \mathcal{D}_\square}{\Gamma \vdash_+ \square_i^\alpha : \square_{i+1}^\beta}\ (\textsc{Sort}^+)$$

$$\frac{(x, A) \in \Gamma \qquad \Gamma \vdash_+}{\Gamma \vdash_+ x : A}\ (\textsc{Var}^+) \qquad \frac{\Gamma \vdash_+ A : \square_i^\gamma \qquad x \notin \mathrm{Var}(\Gamma)}{\Gamma,\ x : A \vdash_+}\ (\textsc{Context}^+)$$

$$\frac{\Gamma \vdash_+ M : \Pi x : A.\, B \qquad \Gamma \vdash_+ N : A}{\Gamma \vdash_+ M\, N : B[x := N]}\ (\textsc{App}^+)$$

$$\frac{\Gamma, x : A \vdash_+ M : B}{\Gamma \vdash_+ \lambda x : A.\, M : \Pi x : A.\, B}\ (\textsc{Lam}^+)$$

$$\frac{\Gamma \vdash_+ A : \square_i^\alpha \qquad \Gamma \vdash_+ B : \square_i^\beta \qquad \mathcal{D}_\to(\gamma) = (\alpha, \beta)}{\Gamma \vdash_+ A \to B : \square_i^\gamma}\ (\textsc{Arrow}^+)$$

$$\frac{\Gamma \vdash_+ A : \square_i^\alpha \qquad \Gamma, x : A \vdash_+ B : \square_i^\beta \qquad \mathcal{D}_\Pi(\gamma) = (\alpha, \beta)}{\Gamma \vdash_+ \Pi x : A.\, B : \square_i^\gamma}\ (\textsc{Pi}^+)$$

Figure 10.4: Typing rules for CC_ω^+

10.4 The Trocq calculus

The final stage of the announced generalisation consists in building an analogue to the parametricity translations available in pure type systems, but for the annotated type theory of § 10.3. This analogue is geared towards proof transfer, and therefore designed to *synthesise* the output of the translation from its input, rather than to *check* that certain pairs of terms are in relation. However, splitting up the interpretation of universes into a lattice of possible relation structures means that the source term of the translation is not enough to characterise the desired output: the translation needs to be informed with some extra information about the expected outcome of the translation. In the Trocq calculus, this extra information is a type of CC_ω^+.

We thus define Trocq *contexts* as lists of quadruples:

$$\Delta ::= \varepsilon \mid \Delta,\ x \,@\, A \sim x' \because x_R \quad \text{where } A \in \mathcal{T}_{CC_\omega^+}$$

We also introduce a conversion function γ from Trocq contexts to CC_ω^+ contexts:

$$\begin{aligned} \gamma(\varepsilon) &= \varepsilon \\ \gamma(\Delta,\ x \,@\, A \sim x' \because x_R) &= \gamma(\Delta),\ x : A \end{aligned}$$

Now, a Trocq judgment is a 4-ary relation, denoted and read in the following way:

$$\Delta \vdash_t M \,@\, A \sim M' \because M_R$$

In context Δ, term M of annotated type A translates to M', because M_R.

Trocq judgments are defined by the rules of Figure 10.5. This definition involves a weakening function for parametricity witnesses, defined as follows.

$$\frac{(\alpha, \beta) \in \mathcal{D}_\square}{\Delta \vdash_t \square_i^\alpha \,@\, \square_{i+1}^\beta \sim \square_i^\alpha \because p_{\square_i}^{\alpha,\beta}} \text{(TrocqSort)}$$

$$\frac{(x, A, x', x_R) \in \Delta \qquad \gamma(\Delta) \vdash_+}{\Delta \vdash_t x \,@\, A \sim x' \because x_R} \text{(TrocqVar)}$$

$$\frac{\Delta \vdash_t M \,@\, \Pi x : A.\, B \sim M' \because M_R \qquad \Delta \vdash_t N \,@\, A \sim N' \because N_R}{\Delta \vdash_t M\, N \,@\, B[x := N] \sim M'\, N' \because M_R\, N\, N'\, N_R} \text{(TrocqApp)}$$

$$\frac{\Delta \vdash_t A \,@\, \square_i^\alpha \sim A' \because A_R \qquad \Delta, x \,@\, A \sim x' \because x_R \vdash_t M \,@\, B \sim M' \because M_R}{\Delta \vdash_t \lambda x : A.\, M \,@\, \Pi x : A.\, B \sim \lambda x' : A'.\, M' \because \lambda x\, x'\, x_R.\, M_R} \text{(TrocqLam)}$$

$$\frac{\Delta \vdash_t A \,@\, \square_i^\alpha \sim A' \because A_R \qquad \Delta \vdash_t B \,@\, \square_i^\beta \sim B' \because B_R \qquad (\alpha, \beta) = \mathcal{D}_\to(\delta)}{\Delta \vdash_t A \to B \,@\, \square_i^\delta \sim A' \to B' \because p_\to^\delta\, A_R\, B_R} \text{(TrocqArrow)}$$

$$\frac{\begin{array}{c}\Delta \vdash_t A \,@\, \square_i^\alpha \sim A' \because A_R \\ \Delta, x \,@\, A \sim x' \because x_R \vdash_t B \,@\, \square_i^\beta \sim B' \because B_R \qquad (\alpha, \beta) = \mathcal{D}_\Pi(\delta)\end{array}}{\Delta \vdash_t \Pi x : A.\, B \,@\, \square_i^\delta \sim \Pi x' : A'.\, B' \because p_\Pi^\delta\, A_R\, B_R} \text{(TrocqPi)}$$

$$\frac{\Delta \vdash_t M \,@\, A \sim M' \because M_R \qquad \gamma(\Delta) \vdash_+ A \preccurlyeq B}{\Delta \vdash_t M \,@\, B \sim M' \because \Downarrow_B^A M_R} \text{(TrocqConv)}$$

Figure 10.5: Trocq rules

$$\Downarrow_{\square_i^{\alpha'}}^{\square_i^\alpha} t_R := \downarrow\downarrow_{\alpha'}^\alpha t_R \qquad \Downarrow_{A'\, M'}^{A\, M} N_R := \Downarrow_{A'}^{A} M\, M'\, N_R$$

$$\Downarrow_{\lambda x : A'.\, B'}^{\lambda x : A.\, B} M\, M'\, N_R := \Downarrow_{B'[x := M']}^{B[x := M]} N_R$$

$$\Downarrow_{\Pi x : A'.\, B'}^{\Pi x : A.\, B} M_R := \lambda x\, x'\, x_R.\, \Downarrow_{B'}^{B} (M_R\, x\, x'\, (\Downarrow_A^{A'} x_R)) \qquad \Downarrow_{A'}^{A} M_R := M_R$$

Figure 10.6: Weakening of parametricity witnesses

Definition 10.4.1
For all $p, q \in \{0, 1, 2_a, 2_b, 3, 4\}$, such that $p \geq q$, we define the unilateral weakening map $\downarrow_q^p : \mathsf{M}_p \to \mathsf{M}_q$ to be the function forgetting the fields from M_p that are not in M_q. For all $\alpha, \beta \in \mathcal{A}$, such that $\alpha \geq \beta$, function $\downarrow\downarrow_\beta^\alpha$ weakening a $\mathrm{Param}^\alpha\, A\, B$ to a $\mathrm{Param}^\beta\, A\, B$ is defined by:

$$\downarrow\downarrow_{(p,q)}^{(m,n)} (R\, ;\, (M^\to, M^\leftarrow)) \quad := \quad (R\, ;\, (\downarrow_p^m M^\to, \downarrow_q^n M^\leftarrow)).$$

The weakening function on parametricity witnesses is defined in Figure 10.6 by extending function $\downarrow\downarrow_\beta^\alpha$ to all relevant pairs of types of CC_ω^+, *i. e.*, $\Downarrow_U^T$ is defined for $T, U \in \mathcal{T}_{CC_\omega^+}$ as soon as $T \preccurlyeq U$.

An associated abstraction theorem relates well-formed Trocq judgments and typing in CC_ω^+ :

Theorem 10.4.2 (Trocq abstraction theorem)

$$\frac{\begin{array}{cc}\gamma(\Delta)\vdash_{+} & \gamma(\Delta)\vdash_{+} M : A \\ \Delta\vdash_t M @ A \sim M' \because M_R & \Delta\vdash_t A @ \square^{\alpha}_i \sim A' \because A_R\end{array}}{\gamma(\Delta)\vdash_{+} M' : A' \quad \text{and} \quad \gamma(\Delta)\vdash_{+} M_R : \mathsf{rel}(A_R)\ M\ M'}$$

Proof. By induction on derivation $\Delta \vdash_t M @ A \sim M' \because M_R$. ■

Note that type A in the typing hypothesis $\gamma(\Delta) \vdash_{+} M : A$ of the abstraction theorem is exactly the extra information passed to the translation. The latter can thus also be seen as an inference algorithm, that infers annotations for the output of the translation from that of the input.

Remark 10.4.3

Since by definition of $p_{\square}^{\alpha,\beta}$ (Equation 9.1), we have $\vdash_t \square^{\alpha} @ \square^{\beta} \sim \square^{\alpha} \because p_{\square}^{\alpha,\beta}$, by applying Theorem 10.4.2 with $\gamma(\Delta) \vdash_{+} A : \square^{\alpha}$, we get:

$$\frac{\gamma(\Delta)\vdash_{+} A : \square^{\alpha} \qquad \Delta\vdash_t A @ \square^{\alpha} \sim A' \because A_R}{\gamma(\Delta)\vdash_{+} A_R : \mathsf{rel}(p_{\square}^{\alpha,\beta})\ A\ A'}.$$

Now by the same definition, for any $\beta \in \mathcal{A}$, $\mathsf{rel}(p_{\square}^{\alpha,\beta}) = \mathrm{Param}^{\alpha}$, hence $\gamma(\Delta) \vdash A_R : \mathrm{Param}^{\alpha}\ A\ A'$, as expected by the type annotation $A : \square^{\alpha}$ in the input of the translation.

Remark 10.4.4

By applying Remark 10.4.3 with $\vdash_{+} \square^{\alpha} : \square^{\beta}$, we get:

$$\vdash_{+} p_{\square}^{\alpha,\beta} : \mathrm{Param}^{\beta}\ \square^{\alpha}\ \square^{\alpha}$$

as expected, provided that $(\alpha, \beta) \in \mathcal{D}_{\square}$.

10.5 Constants

Concrete applications require extending Trocq with constants. Constants are similar to variables, except that they are stored in a global context instead of a typing context. A crucial difference though is that a constant may be assigned several different annotated types in CC_{ω}^{+}. Consider for example, a constant list, standing for the type of polymorphic lists. As $\mathrm{list}\ A$ is the type of lists with elements of type A, it can be annotated with type $\square^{\alpha} \to \square^{\alpha}$ for any $\alpha \in \mathcal{A}$.

Every constant c declared in the global environment has an associated collection of possible annotated types $T_c \subset \mathcal{T}_{CC_{\omega}^{+}}$. We require that all the possible annotated types of a same constant share the same erasure[2] in CC_{ω}, *i. e.*:

2: It is a function $|\cdot|^{-}$ defined as the recursive withdrawal of all annotations on universes.

$$\forall c, \forall A, \forall B, \quad A, B \in T_c \implies |A|^{-} = |B|^{-}$$

For example, $T_{\mathrm{list}} = \{\square^{\alpha} \to \square^{\alpha} \mid \alpha \in \mathcal{A}\}$.

In addition, we provide translations $\mathcal{D}_c(A)$ for each possible annotated type A of each constant c in the global context. For example, $\mathcal{D}_{\mathrm{list}}(\square^{(1,0)} \to \square^{(1,0)})$ is well defined and equal to the following translation:

$$\left(\mathrm{list}, \quad \lambda A\ A'\ A_R.\ (\ \mathrm{List.All2}\ A_R\ ;\ \mathrm{List.map}\ \mathrm{map}(A_R)\ )\right)$$

where $\mathrm{List.All2}\, A_R$ relates lists that are related by A_R element-wise, $\mathrm{List.map}$ is the standard map function on lists and $\mathrm{map}(A_R) : A \to A'$ extracts the function from witness A_R of type $\mathrm{Param}^{(1,0)}\, A\, A' \equiv \Sigma R.\, A \to A'$. Part of these translations can be generated automatically by weakening.

We describe in Figure 10.7 the additional rules for constants in CC_ω^+ and Trocq. Note that for an input term featuring constants, an unfortunate choice of annotation may lead to a stuck translation.

$$\frac{c \in \mathcal{C} \qquad A \in T_c}{\Gamma \vdash c : A}\ (\textsc{Const}^+) \qquad\qquad \frac{\mathcal{D}_c(A) = (c', c_R)}{\Delta \vdash c\,@\,A \sim c' \because c_R}\ (\textsc{TrocqConst})$$

Figure 10.7: Additional constant rules for CC_ω^+ and Trocq

Conclusion and perspectives 11

The functionalities of the prototype plugin presented in this part III can be extended in several directions. It would be particularly fruitful to connect it with tools able to automate the generation of equivalence proofs, such as Pumpkin Pi [68]. Other improvements, *e.g.*, addressing the case of Coq's impredicative sort, involve non-trivial implementation issues, related to Coq's management of universe polymorphism. We now discuss how the current state of this prototype compares with other implemented approaches to proof transfer in interactive theorem proving, listed in chronological order in the summary Table 11.1. For each such tool, the table indicates whether a given feature is available (green), not available (dark orange) or only partially available (yellow).

[68]: Ringer *et al.* (2021), "Proof repair across type equivalences"

In the context of type theory, the idea that the computational content of type isomorphisms can be used for proof transfer already appears in [69]. The first implementation report of a tool based on this idea appeared soon after [70]. Implemented in a meta-language and based on proof rewriting, this heuristic translation was producing a candidate proof term from a given proof term, with no formal guarantee, not even that of being well typed. Generalised rewriting [43], that generalises setoid rewriting to preorders, is also a variant of proof transfer, albeit within the same type. As such, it allows in particular rewriting under binders. The restriction to homogeneous relations however excludes applications to quasi partial equivalence relations (QPER) [71], or to data type representation change.

[69]: Barthe *et al.* (2001), "Type Isomorphisms and Proof Reuse in Dependent Type Theory"

[70]: Magaud (2003), "Changing Data Representation within the Coq System"

[71]: Krishnaswami *et al.* (2013), "Internalizing Relational Parametricity in the Extensional Calculus of Constructions"

The other proof transfer methods we are aware of all address the case of heterogeneous relations. Incidentally, they can thus also be used for the homogeneous case, although this special case is seldom emphasised. The Coq Effective Algebra Library (CoqEAL) [45, 72] and the Isabelle/HOL transfer [73–76] packages pioneered the use of parametricity-based methods for proof transfer, motivated by the refinement of proof-oriented data-structures to computation-oriented counterparts. Together with a subsequent generalisation of the CoqEAL approach [77], these tools address the case of a transfer between a subtype of a certain type A and a quotient of a certain type B, *i. e.*, the case of trivial QPER in which the zig-zag morphism is a partial surjection from A to B.

[73]: Lammich (2013), "Automatic Data Refinement"

[74]: Haftmann *et al.* (2013), "Data Refinement in Isabelle/HOL"

[75]: Huffman *et al.* (2013), "Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL"

[76]: Lammich *et al.* (2019), "Automatic Refinement to Efficient Data Structures: A Comparison of Two Approaches"

[77]: Zimmermann *et al.* (2015), "Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant"

The next two columns of the table concern proof transfer in presence of the univalence principle, either axiomatic in the case of univalent parametricity [14], or computational in the case of [78]. Key ingredients of univalent parametricity were already present in earlier seemingly unpublished work [79], implemented using an outdated ancestor of the MetaCoq library [46].

[78]: Angiuli *et al.* (2021), "Internalizing representation independence with univalence"

[79]: Anand *et al.* (2017), *Revisiting Parametricity: Inductives and Uniformity of Propositions*

Table 11.1 indicates which tools can transfer along *heterogeneous relations*, as this is a prerequisite to changing type representation, and which ones operate by proving an *internal* implication lemma, as opposed to a monolithic translation of an input proof term. We borrow the terminology used in [14], in which *anticipation* refers to the need to define a dedicated structure for the signature to be transported. *Binders* can prevent transfer, as well as *dependent types*. The latter are recovered in presence of univalence. The first published publication [80] on the univalent parametricity translation suggested that the translation does not pull the axiom in when translating terms in the F^ω fragment. However, Trocq can get rid of it for a strictly larger class of terms. Finally, the table indicates which

[80]: Tabareau *et al.* (2018), "Equivalences for free: univalent parametricity for effective transport"

	[Magaud 2003]	Setoid rewrite [Sozeau 2009]	CoqEAL [Cohen et al. 2013]	Isabelle/HOL Transfer (2013)	[Zimmermann and Herbelin 2015]	[Tabareau et al. 2021]	[Angiuli et al. 2021]	Trakt [Blot et al. 2023]	Trocq (2023)
Heterogeneous relations	green	red	green	green	green	green	green	green	green
Internal	red	green	green	green	green	green	green	green	green
No anticipation	green	green	green	green	green	green	red	green	green
Substitution under Π	green	green	red	green	green	green	green	green	green
Substitution in dep. types	green	red	red	red	red	green	green	red	green
No systematic univalence	green	green	green	green	green	red	red	green	green
Preorder relations	red	green	?	?	?	red	?	?	orange
Subrelations	red	green	red	red	red	red	red	red	orange
QERs	red	orange	orange	orange	orange	red	green	red	orange
Subtyping relations	red	red	orange	orange	orange	red	red	orange	orange
System	Coq	Coq	Coq	Isabelle/HOL	Coq	Coq/HoTT	(Cubical) Agda	Coq	Coq or Coq/HoTT

Figure 11.1: Comparison of proof transfer automation devices

approaches can deal with *quasi-equivalence relations* (QER), and with (explicit) subtyping relations.

In its current state, the Trocq plugin can already address the proof transfer bureaucracy of state-of-the-art formal proofs, in the context of abstract mathematics, programme verification, or both [81]. We expect that our work, once put in production, makes it possible to have the same lemma applicable to a wide variety of different types: isomorphic types, subtypes, and quotient types. This framework moreover opens the way to a broader range of extensions, *e. g.*, performing unification modulo both generalised rewriting and heterogeneous transfer relations, potentially solving problems sometimes referred to as *concept alignment*. We conclude with two concrete sticky issues in interactive theorem proving that such extensions could help addressing. The first one is the identification of canonical natural number objects in types, *e. g.*, $\{x : \mathbb{R} \mid \exists n : \mathbb{N}, x = \iota(n)\}$, *etc.* The second one is the identification of different parametric constructions, which happen to *coincide* for some specific classes of parameters, *e. g.*, the ring $\mathbb{Z}/q\mathbb{Z}$, defined for all integers $q > 0$, and the Galois field $\mathbb{F}_q$, defined when $q = p^k$, happen to be canonically isomorphic if and only if q is prime.

[81]: Allamigeon *et al.* (2023), “A Formal Disproof of Hirsch Conjecture”

Implementation of preprocessing tools with Coq-Elpi

Introduction

In the two previous parts, we presented two solutions to deal with the general problem of proof transfer, from a particular angle. The first one, Trakt, was designed starting from `zify`, an *ad hoc* preprocessing tool targeting the `lia` tactic. The second one, Trocq, is a parametricity translation, a more general method with a more theoretical approach. In this part, we focus on the implementation of these tools. In this first chapter, we stand at the level of the software architecture used to create these plugins. In the next chapter, we detail the problems that arise when implementing a parametricity plugin such as Trocq.

Software architecture of a preprocessing plugin 12

12.1 User knowledge base . . . 103
12.1.1 Use of Coq-Elpi databases . 103
12.1.2 Storage of Coq terms 104
12.2 Traversal of the initial goal 104
12.2.1 A translation tactic in Coq-Elpi 104
12.2.2 Trakt: lessons of a first attempt 105

Despite their different goal preprocessing strategies, both tools designed in this thesis tackle a similar problem and have similar needs. Therefore, we devised a common software architecture for both plugins, illustrated in Figure 12.1[1] and presented in this chapter. First, the preprocessing tool needs to know which modifications it needs to apply to the initial goal. The user therefore needs to communicate information to it before entering proof mode. To do this, our architecture integrates a knowledge base into the tool, with commands to add data to it (§ 12.1). Second, the plugin must implement a preprocessing algorithm accessible from Coq's proof mode. The architecture thus includes a tactic that implements a translation taking as input the initial goal and possibly parameters, in order to build, using the knowledge base, the associated goal after preprocessing along with a proof term justifying the substitution (§ 12.2). After running the preprocessing tool, the user only has to prove the associated goal, ideally with another proof automation tool.

1: The parts in orange correspond to the part of the proof scenario to implement in the preprocessing plugin.

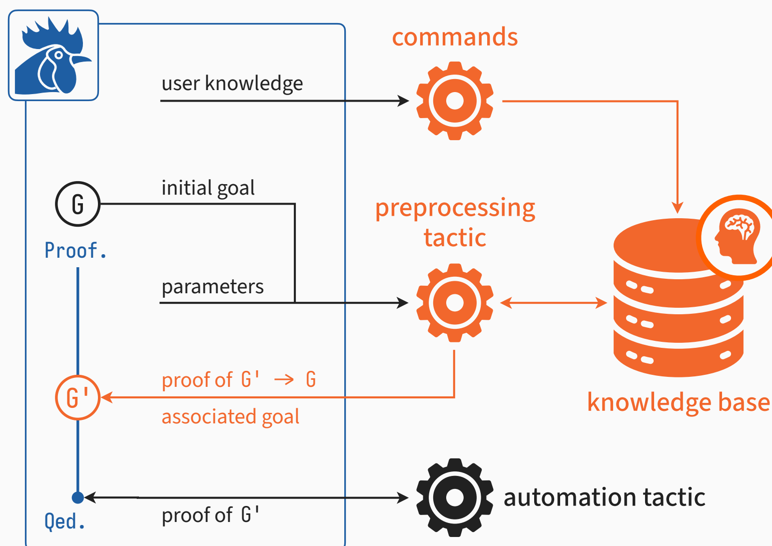


Figure 12.1: Global software architecture for preprocessing plugins developed during this thesis

For example, to preprocess the following initial goal, both in the context of Trakt and Trocq, the user must state how they wish to translate type `int`, its `0` constant and `-` operation, as well as equality:

```
forall (x y : int), x - y = 0
```

This information is user knowledge, given *via* commands and stored in the knowledge base. The user then executes the tactic, providing parameters specific to each tool. The tactic traverses the initial goal and builds the associated goal along with an implication proof between the two, allowing a change in the proof context, leaving only the associated goal to prove. A plausible associated goal could be the following, for instance with `lia` as the automation tactic, associating `int` to `Z`, equality to itself, and the various values in `int` with their counterparts in `Z`:

```
forall (x y : Z), x - y = 0
```

Since both plugins were developed in Coq-Elpi, we will use examples alternatively in Trakt and Trocq to illustrate our points. This technical choice was made for several reasons. First, this meta-language offers a high level of abstraction regarding the syntax of Coq terms. Indeed, the HOAS encoding of terms works very well with eigenvariables of the Elpilanguage, allowing the handling of λ-terms containing binders without ever having to maintain correct De Bruijn indices. In addition, the use of meta-level variables to represent Coq unification variables makes it easy to handle terms with holes — a frequent situation when quoting terms written in Coq's surface syntax — or even to forge terms with holes in the meta-programs and delegate some of the work to Coq by exploiting its elaborator and typechecker. Next, the wrapper formed by Coq-Elpi around the Elpi language provides a real toolbox for meta-programming in Coq, with various useful APIs: creation of commands and tactics, fine control over the behaviour of unification and conversion, definition of terms and types, manipulation of universes, *etc.* Finally, the Elpi language is built around the paradigm of logic programming, lending itself very well to the implementation of recursive algorithms traversing syntax trees, such as those developed during this thesis. *A fortiori*, when they are based on inference rules, like the algorithm at the heart of Trocq, we obtain a direct correspondence between the code and the description of the algorithm on paper.

12.1 User knowledge base

An asset of both plugins presented in this thesis is to be extensible, *i. e.*, to allow the user to customise preprocessing by registering additional information before translating the goal. In the case of Trakt, the additional data are embeddings of types, relations, symbols, *etc.* In the case of Trocq, they are parametricity witnesses. This section presents the use of Coq-Elpi databases as a user knowledge base (§ 12.1.1), as well as interesting technical aspects of the storage of Coq terms in a meta-level database (§ 12.1.2).

12.1.1 Use of Coq-Elpi databases

Plugins Trakt and Trocq use several databases to organise the information to be stored. As explained in § 4.2.1, within Coq-Elpi, a database is a series of facts, *i. e.*, predicate instances that are always true, whose arguments are the data we wish to store. Thus, for example, Trakt registers symbol embeddings as instances of a `symbol` predicate taking as arguments the necessary data, listed in § 6.1.3. In the same way, each different type of information to store is associated with a particular predicate in the database, so as to structure the code and give informative error messages when a piece of data is missing. In Trocq, axioms also have a dedicated predicate, so that the plugin can be used without adding them automatically, and so that goals that do not require them can still be translated. Commands are defined in order to add data in a cleaner way. For example, in the case of a witness `u` of the univalence axiom, the command to use in Trocq is the following:

```
Param Register Univalence u.
```

12.1.2 Storage of Coq terms

Most proof automation tools work with a knowledge base containing constants selected from the context. For example, the `ring` tactic uses a base of instances of an algebraic ring structure, and the `auto` tactic performs a proof search from a list of lemmas chosen by the user. The design of these knowledge bases is difficult because storing Coq terms at the meta level raises subtle questions. Indeed, the surface syntax of Coq hides various pieces of information through notations and implicit elements — arguments, universe instances, *etc.* However, these holes left in the terms are actually variables that can interact with the global state of the proof assistant, *via* unification constraints or the graph of universe constraints for example, as soon as unification is triggered between a term with holes and another term, which happens systematically when reading a database indexed by such terms. This means that storing terms with holes in a database is not a robust choice. However, this syntactic lightness is crucial from the user's point of view, as manually annotating all the terms in their entirety would make the use of the proof assistant very tedious. In cases where the holes left in the terms can be inferred from the rest of the term, we can delegate this work to the Coq elaborator and store the complete term obtained. However, the elaborator always fills a term in a certain context, that may vary between the time at which the knowledge base is filled and the time at which the preprocessing tactic is executed. For instance, two terms that appear to be unifiable may not be so in their complete form, if the term searched in the database is syntactically different from the one that was previously registered after going through the elaborator. Making systematic use of elaboration on the arguments of a Coq command intended to register terms in a meta-level database is therefore a naive solution, which leads to errors that are difficult to trace in the long run. In practice, an interesting trade-off, in order to allow the user a certain amount of freedom in the syntax while storing terms with maximum independence from the global state of Coq, is to store in the database only global references, *i. e.*, terms identified by their name registered in Coq.[2]

2: These include constants, inductive types, and constructors.

12.2 Traversal of the initial goal

Once the knowledge base has been designed and filled by the user, the preprocessing tool translates the initial goal. This translation takes the shape of a recursive algorithm defined by induction on the syntax. This section presents the general structure of a tactic implementing such a translation (§ 12.2.1), as well as the lessons to be learnt from the implementation of Trakt (§ 12.2.2).

12.2.1 A translation tactic in Coq-Elpi

The role of the translation tactic is to traverse the initial goal in order to build both the associated goal and a proof that substituting it for the initial goal is a valid operation. Then, it must apply this proof, to leave the user with a proof context containing only the associated goal, without any additional proof obligation.

Structure of the tactic In both Trakt and Trocq, the translation tactic reads the input goal and calls the main goal traversal predicate, that outputs the associated goal along with a preprocessing proof. We then use the `refine` operation, that

performs essentially the same action as the Coq tactic with the same name, *i. e.*, applying a proof term with holes, the holes representing the new proof obligations. In our case, we expect a single hole, having the type of the associated goal and ideally being provable using a proof automation tactic. The call is therefore the following:

```
refine {{ lp:Proof (_ : lp:EndGoalTy) }} InitialGoal NewGoals
```

Variable `Proof` contains the proof of implication, and `EndGoalTy` is the associated goal, both generated by the translation.

Recursive translation predicate Implementing the translation algorithm is the focal point of a preprocessing tactic. It is a recursive predicate that takes as input at least one term that in the first call will be the initial goal, and returns as output at least two terms, the associated goal and the proof of validity for the substitution.

Elpi predicates can be defined with several instances, and Prolog-like unification in this meta-language allows selecting the instance corresponding to the current case to be processed. If the arguments in the head of the instance do not unify with the arguments of the current call to the predicate, the execution moves on to the next instance, and so on until failure.[3] In the case of a predicate defined by induction on the syntax of a Coq term, one writes at least one instance per construct available in the language. Several sub-cases for a same construct can be distinguished by executing test predicates at the beginning of the body of each instance. If one of these predicates fails, the next instance will be selected. To force the program to explore only one instance, one can add a *cut* — with character `!` — after these test predicates. All this makes it easier to organise the code and define the order of priority when testing the various available cases.

3: It is good practice to always define a last instance catching all the remaining cases and failing with an error message.

Furthermore, in order to apply in Coq a proof term as complete as possible when calling `refine`, it is necessary to give the proof to a typing and/or elaboration predicate in order to fill the last holes that do not represent a future proof but a type annotation left implicit in the translation predicate. This allows delegating some of the work to Coq and writing the proofs in a more natural way during the development of the plugin.

12.2.2 Trakt: lessons of a first attempt

The first prototype implemented following the software architecture presented earlier is Trakt, presented in this thesis in part I. The plugin improves preprocessing based on canonisation, previously embodied by the `zify` tactic, by performing a similar but extended translation to handle goals in the SMT family. The implementation of Trakt allowed to identify interesting design patterns to bear in mind for future projects in Coq-Elpi. It has a few flaws, but overall it achieves its objective, as shown by its successful integration into the SMTCoq library. This subsection is an experience report.

Encoding of terms in HOAS A first useful feature of Coq-Elpi is the encoding of terms in HOAS. Thanks to this encoding, contexts can be expressed as functions in the meta-language. For example, a context $C[\cdot]$ is represented with a variable `C` of type `term -> term`. Completing the context with a term `x` then amounts to performing a functional application `C x`; updating it by going under a new node,

for example a one-argument function `f`, is done by creating a new meta-function `C'`:[4]

```
pi x\ C' x = C (app [f, x])
```

In this encoding, binders are the association of a term representing the type of the bound variable and a meta-function representing the body of the function in the case of an abstraction or the codomain in the case of a dependent product. Crossing such a binder is done by creating a fresh local variable using the same process as above. All the terms computed from the application of the meta-function are expressed as a function of this variable, allowing terms to remain closed throughout the process. As these terms are also meta-functions, it suffices to add a binder constructor to obtain a well-formed Coq term. For example, here is a snippet extracted from the instance of the main predicate of Trakt in charge of preprocessing dependent products:[5]

```
preprocess (prod N T F) /* ... */ :- !,
  @pi-decl N T x\ preprocess (F x) /* ... */ (F' x) (PF x),
  % ...
```

In one line of code, we introduce a fresh variable `x` in the domain `T` to make a recursive call on the codomain `F x` and get the associated codomain `F' x` as well as a proof `PF x`. The rest of the predicate reworks these terms to obtain the associated dependent product and the preprocessing proof.

Finally, the abstraction of a subterm t in a term u into a function f such that $u \equiv f\,t$ is a simple task in Coq-Elpi. Indeed, there is a `copy` predicate in the standard library, initially defined as a deep identity: it traverses the whole structure of a term and applies an identity to the leaves of the tree. By adding a local instance of `copy`, it is possible to perform substitutions: with the additional instance `copy X Y`, any occurrence of `X` encountered during the traversal is replaced with `Y` in the output term of the predicate. By representing t with a variable `T` and u with `U`, we can abstract t in u in a single line:

```
pi x\ copy T x => copy U (F x).
```

We replace all the occurrences of `T` in `U` with a fresh variable `x`, yielding a meta-function `F` that represents the desired abstraction. One can then use this meta-function directly or turn it into a proper Coq function by adding a binder on top.

4: In this code, `x` is a universal constant representing a bound variable, allowing access to the body of meta-function `C` to define `C'`. Because of its status in Elpi, it must be made explicit as an argument to `C'`, so that it does not escape its local scope.

5: In Coq-Elpi, dependent products are represented with `prod N T F` terms, where `N` is the display name of the bound variable in Coq, `T` is the domain, and `F` is a meta-function containing the codomain.

Customisation of Coq terms Another crucial feature in Coq-Elpi is that it allows developers to define new types and add constants with the type of their choice. It is therefore possible to emulate the behaviour of algebraic types found in more traditional functional languages, by declaring a new type and various constants to represent its constructors. Here is an example of definition of natural numbers in Elpi:

```
kind nat type.
type zero nat.
type succ nat -> nat.
```

As these values are not proper algebraic types, the set of constructors is not closed and the developer or user of an Elpi codebase can very well add new constructors to a type defined inside it. For instance, the Coq-Elpi API for Coq term manipulation exposes an *extensible* representation of terms, which is excellent news for meta-programming. Indeed, it is possible to create new nodes in the AST of Coq terms to represent various useful pieces of information when handling terms at

the meta level. This technique is used in Trakt, where two new constructors are added, for the purpose of annotating Coq terms:

```
type prod2 name → term → term → (term → term) → term.
type cast term → term.
```

The prod2 constructor is similar to the original prod constructor used to represent dependent products, the difference being that it contains an additional argument of type term. Thanks to this argument, it is possible to make a dependent product containing information about the domain before and after translation. This piece of information can be used to find out if the type of a bound variable has changed during translation, so that the proof can be adapted accordingly. The cast constructor is used to distinguish embedding functions added by the translation algorithm from any embedding functions that might already be present in the initial goal before translation. In both cases, a clean-up procedure must be executed after this information has been used, so that the term returned to Coq can be translated again in the proof assistant's native syntax. In fact, in order to make Coq terms available in the meta-language, the Elpi type used to represent them is in bijection with the native type in Coq, so the addition of these new constructors prevents Coq-Elpi from switching between both representations. In the case of Trakt, this clean-up procedure is straightforward, since the only thing to do is to delete cast annotations, leaving the underlying term instead, and forget one of the two domains in the prod2 nodes, replacing them with prod nodes, in order to recover a well-formed Coq term.

Reification of proofs A useful meta-programming pattern identified during the implementation of Trakt is the reification of proofs, *i. e.*, the design of an algebraic type in the meta-language to represent proofs. This gives a higher level of abstraction than handling the raw Coq terms representing the various proof steps to perform in order to rewrite the initial goal into the associated goal. In addition, the use of these reified proofs helps to understand the proof fragments built by the meta-program during the translation, and allows externalising the generation of the final Coq proof term into another predicate than the main one, thus improving readability in the code of the plugin. For instance, proofs within Trakt are first generated in an Elpi type proof whose constructors represent the various possible proof steps during the translation, then they go through a function that reconstructs the corresponding Coq proof fragments before returning the final term to the proof assistant.

Traversing the term and maintaining the context Despite all the positive points identified, Trakt suffers from some heaviness, in particular in its main translation predicate. Indeed, as explained in § 6.2, the translation must distinguish covariant positions from contravariant positions, preprocess terms differently depending on whether their type is embeddable or not, and keep the context of the current term in memory in order to generate rewriting proofs applying to the whole logical atom, for proof composition. For all these reasons, the predicate has additional parameters to know which case of the translation to apply to the current node. Each new feature added is a list of new special cases to deal with, which adds more arguments and tests to the predicate.[6] In addition, there is a high number of case analyses that are sometimes deep, which makes them difficult to express just by adding instances to the predicate. They are therefore implemented with conditional branches, which are not a native construction in Elpi. Branches are made *via* an explicit test predicate if, which slightly hinders readability.

6: Here, the problem is not particularly due to the meta-language, but rather to the *ad hoc* design approach of the plugin, targeting short-term utility for Coq users.

Trakt is a success in practice, but its implementation reflects its bottom-up model, centred first and foremost on a concrete problem to solve, with some abstraction giving the plugin its extensibility and flexibility, detailed in § 6. The few limitations mentioned are some of the reasons that led us to think about a more general solution, a top-down design from theory to practice. Parametricity plugins developed in the 2010s hold the promise of gathering all the cases of translation into a general framework, with an implementation that is admittedly more difficult but also more elegant. Trocq pushes this approach towards unifying the various parametricity translations themselves, bringing certain subtleties to the implementation, that are presented in the next chapter.

Implementation of a parametricity plugin 13

13.1 Generating and inhabiting the parametricity hierarchy 110
13.1.1 Generation of the hierarchy and plugin set-up 110
13.1.2 Flexibility of parametricity witnesses 112
13.2 Implementation of the parametricity relation . . . 113
13.2.1 From inference rules to a logical program 113
13.2.2 Useful Coq-Elpi features . . 115
13.3 Parametricity class inference 116
13.3.1 Problem definition 117
13.3.2 Solution chosen in Trocq . 119
13.3.3 Implementation 120
13.3.4 Weakening and subtyping . 122
13.4 Universe polymorphism . . 122
13.4.1 Clearing typical ambiguity . 123
13.4.2 Algebraic universes and bound universes 124

The second prototype designed during this thesis, Trocq, aims to solve the same preprocessing problem as Trakt presented earlier, while being able to translate any CC_ω term. Trocq is in particular a refinement of the univalent parametricity translation, dispensing with the univalence axiom wherever a manual processing would not use it. This refinement involves the design of a hierarchy of Σ-types representing types of more or less rich parametricity witnesses, ranging from the raw parametricity witness to the univalent witness. The levels of witnesses, called parametricity classes, are then constrained during the traversal of the goal to allow only the witnesses that are rich enough to build the preprocessing proof of the initial goal. The final classes are set after this traversal in a separate procedure, in order to obtain a unique translation. Constraints added during the process require a syntactic way to represent and manipulate parametricity classes, so we use an annotated type theory where all universes come with a parametricity class. Finally, in a real-world context, the translation works from a knowledge base containing user-proven parametricity witnesses on different constants that may appear in the goals to be translated.

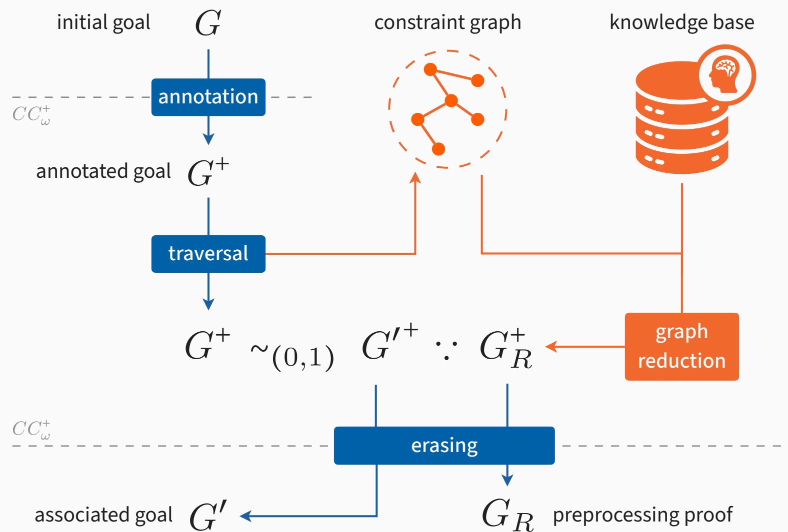


Figure 13.1: Mode of operation of Trocq

Thus, the mode of operation of the implementation of Trocq can be summarised by Figure 13.1. First, the initial goal G is annotated by adding fresh variables on all universes, *i. e.*, variable parametricity classes that are still unconstrained, yielding a goal G^+ that will be the input goal to traverse. This traversal applies structural rules and generates a new goal G'^+ as well as a parametricity witness G_R^+ relating the two goals at level $(0,1)$, the smallest level subsequently allowing extraction of a proof of $G'^+ \rightarrow G^+$ from the witness. The traversal adds various constraints on the parametricity classes, that are represented with a constraint graph. After the traversal, the graph is reduced and the final parametricity classes are set. The assignment of parametricity class variables triggers queries in the knowledge base to retrieve any required user-provided witnesses, at the right level for the overall proof to be well typed. Finally, once all the terms are

complete, erasure is performed to recover an associated goal G' and a preprocessing proof to extract from G_R, as two valid Coq terms.

This chapter raises various points of technical interest in the implementation of Trocq. First, we look at the generation of the parametricity hierarchy, the framework on which everything else in the plugin is based (§ 13.1). Next, we explain how Coq-Elpi's paradigm brings the implementation of this framework closer to its relational description (§ 13.2), and then describe the implementation of parametricity class inference (§ 13.3). Finally, we identify the limitations of the current implementation of universe polymorphism in Coq, a crucial feature in the implementation of Trocq (§ 13.4).

13.1 Generating and inhabiting the parametricity hierarchy

In the implementation of Trocq, we use meta-programming to write the goal traversal procedure, but also to set up the plugin beforehand, to automate away the combinatorial complexity introduced by the parametricity hierarchy. There exist 6 levels in the hierarchy and a parametricity class is a combination of a covariant level and a contravariant level. Therefore, there are in total 36 possible parametricity classes, and consequently as many variants for each definition indexed by a class: parametricity witness types, parametricity lemmas, weakening functions, *etc.* This multiplicity is such that it would be unreasonable to write all the definitions manually. The symmetrical formulation of parametricity witnesses reduces this manual effort to 6 variants, one for each level of the hierarchy. The remaining terms can then be generated by combining the 6 base variants. In this section, we show how this generation using Coq-Elpi helps both the developer when setting up the plugin and the user by handily manipulating the parametricity witnesses they added to the knowledge base.

13.1.1 Generation of the hierarchy and plugin set-up

The theoretical presentation of Trocq defines a hierarchy of parametricity witness types to relate two types A and B, ranging from the raw parametricity witness type $A \to B \to \square$ to the univalent witness type $\mathrm{Param}^{\top} A\, B$ from Theorem 9.1.6. The implementation must define a witness type for every possible level in the hierarchy and define as many versions of the parametricity lemmas.

Parametricity witness types Parametricity witness types $\mathrm{Param}^{(\alpha,\beta)}$, from Definition 9.1.7, are indexed by a parametricity class (α, β) and defined as the combination of a relation with two unilateral witnesses $\mathrm{M}_\alpha\ R$ and $\mathrm{M}_\beta\ R^{-1}$, each one concerning one direction of relation R.

In Coq, dependent pairs can be represented in an equivalent way with *records*. Indeed, records ease the manipulation of structures because they are flat[1] and it is possible to name the projections applied to them.[2] The implementation of Trocq therefore uses the following record as a concrete representation of $\mathrm{Param}^{(\alpha,\beta)}$:

```
Record Param^(α,β)@{i} (A B : Type@{i}) := {
  R : A → B → Type@{i};
  covariant : Map_α R;
```

1: All terms at the same level in the structure can be obtained with the same number of projections.

2: They are the *fields* of the record.

```
  contravariant : Map_β (sym_rel R)
}.
```

In this family of records, the implementation of the M unilateral witness types is faithful to the theoretical description, by simply reformulating dependent pairs as records and naming the fields. Thus, the M_4 unilateral univalent witness type is implemented by the following record:

```
Record Map4@{i} {A B : Type@{i}} (R : A → B → Type@{i}) := {
  map : A → B;
  map_in_R : forall (a : A) (b : B), map a = b → R a b;
  R_in_map : forall (a : A) (b : B), R a b → map a = b;
  R_in_mapK : forall (a : A) (b : B) (r : R a b),
                (map_in_R a b (R_in_map a b r)) = r
}.
```

The `map` function is indeed a map from `A` to `B`; the field `map_in_R` describes the property for the graph of this map to be included in the relation `R`; the opposite property, *i. e.*, that relation `R` is included in the graph of `map`, is described by the field `R_in_map`; finally, the field `R_in_mapK` indicates that both previous fields cancel each other out.[3] To obtain the remaining unilateral witness types, it suffices to remove fields from this record. For example, the record type without the last field corresponds to unilateral witness type M_3 and an empty record type corresponds to M_0. The association of a unilateral witness at level α on a relation `R` and a unilateral witness at level β on the inverse relation `sym_rel R` actually corresponds to a parametricity witness $\mathrm{Param}^{(\alpha,\beta)}$.

3: The naming is inspired by the MathComp library, where a cancellation property is named with a suffix `K` for “cancel”.

All these records can be generated with a Coq-Elpi command, by writing a predicate taking as a parameter the required parametricity class for the witness. This predicate then opens a module as a namespace dedicated to this parametricity class and defines the record. Here’s how to make such a definition in Coq-Elpi:[4]

4: For readability, we do not mention universes at this point.

```
1  coq.env.begin-module "Param43" none,
2  RelDecl =
3    parameter "A" _ {{ Type }} (a\
4    parameter "B" _ {{ Type }} (b\
5      record "Rel" {{ Type }} "BuildRel" (
6        field [] "R" {{ lp:a → lp:b → Type }} (r\
7        field [] "covariant" {{ Map4.Has lp:r }} (_\
8        field [] "contravariant" {{ Map3.Has (sym_rel lp:r) }} (_\
9      end-record)))))),
10 coq.env.add-indt RelDecl _,
11 coq.env.end-module _.
```

Values `parameter`, `record`, `field`, and `end-record` are constructors of a Coq-Elpi type used to represent Coq definitions of inductive types. The `coq.env.*` API contains all the functions to interact with the Coq environment, and in particular to make new definitions.

Parametricity lemmas Among all the possible cases in the parametricity translation in Trocq, some are only a matter of λ-calculus — such as application or abstraction — and building the witness essentially consists in using a combinator to join the results of recursive calls; other cases require proofs defined independently of the goal traversal procedure. In the case of the universe and the dependent product, these are proofs $p_{\square}$ and p_{Π}, which we coined as *parametricity*

lemmas. These lemmas are represented in Coq by a family of terms with the following types:[5]

```
Definition Param□^γ : Param^γ Type Type.

Definition ParamΠ^γ
  (A A' : Type) (A_R : Param^α A A')
  (B : A → Type) (B' : A' → Type)
  (B_R : forall a a' a_R, Param^β (B a) (B' a')) :
    Param^γ (forall (a : A), B a) (forall (a' : A'), B' a').
```

5: In the case of the dependent product, the retained parametricity classes are

$$(\alpha, \beta) = \mathcal{D}_\Pi(\gamma)$$

These definitions are done in the same way as for the records, first making the proofs manually for the 6 levels of the hierarchy, yielding 6 unilateral witnesses to be combined together to obtain the final proofs. A case analysis is performed on parametricity class γ to determine whether the principle of univalence — for the universe — or function extensionality — for the dependent product — is needed to carry out the proof. The difference with parametricity witness types is that the definitions here concern constants and not inductive types. The content of a definition is therefore a Coq term in the encoding of Coq-Elpi, and the predicate used to make the definition is `coq.env.add-const`.

13.1.2 Flexibility of parametricity witnesses

In order to implement the parametricity relation in Trocq, it is necessary not only to define all possible parametricity witness types, but also to ensure a certain degree of compatibility between these types and flexibility in their use. For both the developer and the user, the content of parametricity witnesses must be transparent, and the hierarchy must not involve heaviness either in the code or in the user declarations. For example, it must be easy to extract a piece of information from a record that has a sufficient class to contain it, and it must be possible to accept a witness supplied by the user wherever a weaker witness is expected.

Parametricity witnesses in Trocq can be *weakened* using a function described in Figure 10.6. This weakening occurs when the available witness is richer than the expected witness type, so that the overall parametricity witness remains well typed. This allows the user to declare only one parametricity witness to relate two constants, at the highest parametricity class for which the proof is possible. Thus, wherever a witness is required on these constants at a level reachable by weakening from the level of the provided witness, a weakening function is added automatically by the plugin.

A weakening function can be defined between a source annotated type and a target annotated type if the latter is a subtype of the former. In the base case where the witness is a record, the weakening corresponds to forgetting fields and recomposing the remaining fields in a new weaker record. The various weakenings are generated from manually defined atomic forgetful functions, deleting the highest level field in a unilateral parametricity witness. For example, here are the types of the forgetful function from level 3 to level 2_a and of the forgetful function from class $(4,3)$ to class $(4,2_a)$ generated with Coq-Elpi from the former:

```
Definition forgetMap^3_{2a} {A B : Type} {R : A → B → Type} :
  Map_3 R → Map_{2a} R.

Definition forget^{(4,3)}_{(4,2a)} {A B : Type} :
  Param^{(4,3)} A B → Param^{(4,2a)} A B.
```

Projections In the previous definition of the Map_4 record, the fields have been named to make data extraction more readable. However, these fields are those of a unilateral record that is then included in another record, and each record exists in a separate namespace. In this state, extracting a field is syntactically cumbersome and dependent on the parametricity class of the witness. However, the forgetful functions can be declared as *coercions*, allowing Coq's typechecker to forget an arbitrary number of fields and thus check that a cookie is sufficiently rich. Thanks to this, a projection function is defined once for each field, and this function can be applied to all parametricity witnesses containing this field while preserving typing. For example, here is the projection on field map_in_R corresponding to level 2_a in the hierarchy:

```
Definition map_in_R {A B : Type} :
  Param^(2a,0) A B → forall (a : A) (b : B), map R a = b → R a b.
```

This projection actually concerns the left-to-right unilateral witness, since it applies to a witness of class $(2_a, 0)$, but this method also allows the fields of the right-to-left unilateral witness to be named differently. Thus, Trocq also contains definitions for symmetrical projections. The symmetrical projection of map_in_R is named comap_in_R and has the following type:[6]

```
Definition comap_in_R {A B : Type} :
  Param^(0,2a) A B → forall (b : B) (a : A), comap R b = a → R a b.
```

6: Value comap is the symmetrical field of map and has type A → B.

13.2 Implementation of the parametricity relation

The parametricity framework of Trocq was designed with the aim of being implemented in Coq. Its relational presentation, detailed in Figure 10.5, is heavier than the traditional presentation of parametricity translations, but has the advantage of making explicit details that are important at the time of implementation, namely the steps of manipulation of bound variables as well as the origin of the terms present in the conclusion of the inference rules. To implement Trocq, Coq-Elpi is a natural choice, its logic paradigm being entirely in line with this relational presentation. These various design and implementation choices have resulted in a high degree of similarity between the theoretical presentation of Trocq and the code of the relation. This section highlights this readability in the implementation of Trocq.

13.2.1 From inference rules to a logical program

First of all, we can see that the deductive presentation of an algorithm fits in very well with logic programming. Indeed, the algorithm can be implemented with an Elpi predicate, where each case of the algorithm corresponds to an instance of the predicate, and for each case, the head of the instance corresponds to the conclusion and the body of the instance corresponds to the premises. So, just as we would construct a tree on paper by stacking various rules of the algorithm, starting with the conclusion to be obtained at the root of the tree, the Coq-Elpi implementation consists of a call to the parametricity predicate, with each recursive call representing a new rule to add to a branch of the tree.

However, the theoretical presentation keeps a certain level of abstraction through the presence of the TrocqConv rule. This rule concentrates all the flexibility of parametricity in Trocq, in the fact that it can be added anywhere in the derivation

tree to make valid a parametricity witness stronger than necessary. Although this elegant rule avoids adding weakenings to all the other rules, it makes the goal traversal procedure non-deterministic because there are two possible rules for each construct of the language. It is up to the implementation to choose when to use this TrocqConv rule. Moving from inference rules to the logical program involves making the goal traversal algorithm deterministic, followed by associating each element[7] appearing in the rules with a corresponding atomic code fragment in Coq-Elpi.

7: Values, operations, kinds of premises, *etc.*

Making the algorithm deterministic In Trocq, the implementation of the parametricity relation systematically uses the TrocqConv weakening rule in the base cases, *i. e.*, on variables and constants. These are cases in which a premise might not be provable in the absence of weakening. In the case of variables — the rule TrocqVar —, the premise directly checks the parametricity context Ξ to find an associated witness. However, this witness does not necessarily have an annotated type identical to the one at which the variable is processed. In the case where the type of the variable contains parametricity classes, the weakening rule is even a crucial tool to constrain these classes during traversal of the goal. The case of constants is analogous: the knowledge base contains a finite number of possible associations for the same constant, with annotated types representing precisely the dependencies needed to create the parametricity witnesses, and a constant is not always processed exactly at one of these types. Weakening allows using a potentially richer witness declared in the database if the annotated type desired during the goal traversal does not exactly exist in the database. Using weakening in the other cases of the predicate can lead to computing a richer parametricity witness than necessary, which is contrary to the objective of Trocq.

Another non-deterministic element is visible in the case of the application — rule TrocqApp —, the only case containing a term whose origin is not set: the domain A of the function at the head of the application. Rather than creating a fresh variable at this point and potentially causing an unnecessary weakening in the recursive call on f, the implementation retrieves the type of f from the context to read the right value for A. In this way, the potential weakening operation carried out when processing f concerns type B.

Correspondence between rules and code To study the link between rules and code, let us start with the case of bound variables, using the TrocqVar and TrocqConv rules. Here is the combination of these two rules implemented in Trocq:

$$\frac{(x, T, x', x_R) \in \Delta \qquad \gamma(\Delta) \vdash_{+} T \preccurlyeq T'}{\Delta \vdash_t x @ T' \sim x' \because \Downarrow^{T}_{T'} x_R}$$

Now here is the corresponding instance of the predicate for the parametricity relation in Trocq:[8]

8: In the next few blocks of code, the display predicates have been removed as they are not useful in the presentation and do not detract from our argument.

```
param X T' X' (W XR) :- name X, !,
  param.store X T X' XR,
  annot.sub-type T T',
  weakening T T' (wfun W).
```

The `param` predicate with four arguments is the main parametricity predicate. The head of the instance corresponds to the conclusion of the rule, where `W` is the weakening function generated in line 4. The `name X` condition is used to check that `X` is indeed a variable and to make this instance fail on all the other terms,

in order to execute the instance dedicated to them instead. Lines 2 and 3 are the premises. Predicate `param.store` is used to represent the Δ parametricity context. It therefore appears that the association between the inference rules and the predicate instance is fairly transparent.

Let us now look at a more complex case, that of the arrow type, which involves constraints between parametricity classes as well as recursive calls to `param`:[9]

9: The inference rule is a deliberately reorganised version of the TrocqArrow rule, in which some parametricity classes are explicitly named and universe constraints do not appear. Indeed, as they are delegated to Coq, they do not appear in the code and can be ignored here.

$$\frac{\begin{array}{cc} C = (M, N) & (C_A, C_B) = \mathcal{D}_{\to}(C) \\ C_A = (M_A, N_A) & \Delta \vdash_t A \,@\, \square^{(M_A,N_A)} \sim A' \because A_R \\ C_B = (M_B, N_B) & \Delta \vdash_t B \,@\, \square^{(M_B,N_B)} \sim B' \because B_R \end{array}}{\Delta \vdash_t A \to B \,@\, \square^{(M,N)} \sim A' \to B' \because p^C_{\to}\, A_R\, B_R}$$

```
1  param
2    (prod _ A (_\ B)) (app [pglobal (const PType) _, M, N])
3    (prod `_` A' (_\ B')) (app [pglobal (const ParamArrow) UI|Args]) :-
4      param.db.ptype PType, !, std.do! [
5        cstr.univ-link C M N,
6        cstr.dep-arrow C CA CB,
7        cstr.univ-link CA MA NA,
8        param A (app [pglobal (const PType) _, MA, NA]) A' AR,
9        cstr.univ-link CB MB NB,
10       param B (app [pglobal (const PType) _, MB, NB]) B' BR,
11       param.db.param-arrow C ParamArrow,
12       prune UI [],
13       util.if-suspend C (param-class.requires-axiom C) (
14         coq.univ-instance UI0 [],
15         Args = [
16           pglobal (const {param.db.funext}) UI0, A, A', AR, B, B', BR
17         ]
18       ) (
19         Args = [A, A', AR, B, B', BR]
20       )
21     ].
```

The six premises are represented by lines 5 to 10 in the code block, in the same order as in the inference rule. The remaining code retrieves the $p^C_{\to}$ proof present in the conclusion (variable `ParamArrow`) and applies the right arguments to it, some of which are implicit in the paper presentation.

13.2.2 Useful Coq-Elpi features

As showed in the inference rules describing the parametricity translation of Trocq — Figure 10.5 —, it involves operations of different kinds: recursive calls, constraints on parametricity classes, proof construction from parametricity lemmas or witnesses added by the user. Making all these operations available requires a certain amount of software infrastructure, however hidden in Trocq's implementation behind the division of the code into several files and specific features of Coq-Elpi, such as *goal suspension* or *Constraint Handling Rules* [51] (CHR).

[51]: Frühwirth (1994), "Constraint Handling Rules"

First, splitting the code makes it easier to read. This is why, in the code of Trocq, all the constraint logic on parametricity classes goes through an API of `cstr.*` predicates, so as not to expose this part of the implementation in the definition of the main parametricity predicate, and thus remain as faithful as possible to the inference rules.

Second, the use of Coq-Elpi's CHR allows maintaining a global state containing the various parametricity class constraints throughout the goal traversal. This has the advantage that the data structure storing the constraints is invisible in the `param` predicate,[10] but also that no reified encoding of the parametricity classes is necessary in the annotated terms. Indeed, a central feature in the use of CHR is the reification of variables in the head of the rules. Unlike classic Elpi code modelled on Prolog, where the notion of term comparison is based on unification, in the context of CHR, the first phase in the execution of a rule, pattern matching, is carried out at the meta level. At this level, two syntactically different variables in the various arguments of suspended goals can therefore not be unified. Among other things, this allows indexing data structures on variables, and therefore leaving the parametricity class variables as they are in the annotated terms.[11]

10: Because it is a global state, it is possible to never name or mention it in the code.

11: Internally, they are still associated with an integer, because the data structure storing the constraints requires the keys to be comparable, which is a simpler task on integers than on variables.

Third, goal suspension allows implementing the algorithm in the same way regardless of the status of the initial goal. Indeed, when annotating a goal before traversal, many fresh variables are created to represent parametricity classes that are still unknown. In the initial call to the `param` predicate, in general, the parametricity class at which we wish to process the goal is known: we want class $(0, 1)$ in order to extract a function from the new goal to this initial goal. In the code of the relation for arrow types, variables `M` and `N` are therefore ground, which then determines `C` and the other variables in the program that depend on it. However, as recursive calls are made, it is possible that some parametricity class variables remain undefined, not having a ground value before running a subsequent procedure to determine parametricity classes.[12] Yet, the code may have to compute results from such variables.[13] Elpi's goal suspension feature is therefore welcome in the implementation of Trocq, since it allows blocking a computation on a variable for as long as that variable is undefined, and then to wake it up as soon as a ground parametricity class is assigned to the variable.

12: The associated problem, the solution chosen and its implementation are the topic of the next section.

13: For example, retrieving a value from a database — line 11 in the latest code block — or testing whether the level of a variable requires an axiom — line 13.

13.3 Parametricity class inference

A particularity of the parametricity relation in Trocq is that it can relate a same term to several valid associated terms. This is because the relation starts from a term in CC_ω^+ containing universes annotated with parametricity classes. However, in our case, the input term is obtained by an automatic annotation of the initial goal that creates a fresh variable for each parametricity class. Furthermore, the rules of Trocq do not explicitly constrain these classes to be equal to a particular value, but simply to be above or below a value. Thus, the goal traversal does not *set* the different parametricity classes found in the annotated types, but only *constrains* them. At the end of the process, some parametricity classes may then have several valid solutions.

For example, when translating $\Pi A : \square . A \to A$ at level $(0, 1)$, a parametricity witness of level $(2_\mathrm{a}, 0)$ *at least* is required on subterm $\square$. This means that all entries of the relation for $\square$ that are above this level are also acceptable, as they contain at least the required amount of information. Similarly, processing $\mathbb{N} \to \mathbb{N}$ at level $(1, 0)$ will require two parametricity witnesses on $\mathbb{N}$, one at level $(0, 1)$ *at least* and the other at level $(1, 0)$ *at least*. All witnesses of higher level yield a well-typed final proof.

However, it is important to determine a final value for these classes, as these values have a potential impact on the amount of information requested from the user, and even on the axioms required to perform preprocessing. In order to

avoid the use of axioms as much as possible and request as little information as possible from the user, it seems worthwhile to try to minimise all the parametricity classes in the output witness. However, this minimisation problem does not always have a solution, in particular when the goal contains user constants. The implementation of Trocq uses a heuristic that in many cases allows a satisfactory solution to be obtained.

In this section, we define the inference problem and illustrate it with an example. Next, we explore the various solutions considered and justify the technical choices made in Trocq. Finally, we give details on the implementation of weakening and subtyping, as these are key points for the parametricity class inference to work correctly.

13.3.1 Problem definition

In order to illustrate the problem of parametricity class inference, let us look at an example. We consider a processing of the following initial goal, freshly annotated automatically by Trocq:

$$\Pi F : \square^{\alpha} \to \square^{\beta}.\, \Pi A : \square^{\gamma}.\, F\ A \to F\ A$$

It contains three fresh variables α, β, and γ, one for each universe. The trace of the traversal of this term made by Trocq is the following:

Processing of the initial goal at type $\square^{(0,1)}$.
Application of TrocqPi.

- Processing of domain $\square^{\alpha} \to \square^{\beta}$ at type $\square^{(2_a,0)}$.
  Application of TrocqArrow.
  - Processing of $\square^{\alpha}$ at type $\square^{(0,2_b)}$.
    Application of TrocqSort.
    Addition of constraint $(\alpha, (0, 2_b)) \in \mathcal{D}_{\square}$.
  - Processing of $\square^{\beta}$ at type $\square^{(2_a,0)}$.
    Application of TrocqSort.
    Addition of constraint $(\beta, (2_a, 0)) \in \mathcal{D}_{\square}$.
- Processing of $\Pi A : \square^{\gamma}.\, F\ A \to F\ A$ at type $\square^{(0,1)}$.
  Application of TrocqPi.
  - Processing of $\square^{\gamma}$ at type $\square^{(2_a,0)}$.
    Application of TrocqSort.
    Addition of constraint $(\gamma, (2_a, 0)) \in \mathcal{D}_{\square}$.
  - Processing of $F\ A \to F\ A$ at type $\square^{(0,1)}$.
    Application of TrocqArrow.
    - Processing of $F\ A$ at type $\square^{(1,0)}$.
      Application of TrocqApp taking $\square^{\alpha}$ as the domain of F.
      - Processing of F at type $\square^{\alpha} \to \square^{(1,0)}$.
        Application of TrocqConv then TrocqVar.
        Addition of constraint $\beta \geq (1, 0)$.
      - Processing of A at type $\square^{\alpha}$.
        Application of TrocqConv then TrocqVar.
        Addition of constraint $\gamma \geq \alpha$.
    - Processing of $F\ A$ at type $\square^{(0,1)}$.
      Application of TrocqApp taking $\square^{\alpha}$ as the domain of F.

- Processing of F at type $\square^\alpha \to \square^{(0,1)}$.
  Application of TrocqConv then TrocqVar.
  Addition of constraint $\beta \geq (0,1)$.
- Processing of A at type $\square^\alpha$.
  Application of TrocqConv then TrocqVar.
  Addition of constraint $\gamma \geq \alpha$ — duplicate.

We therefore get a set of constraints at the end of the traversal, describing sets of admissible values for α, β, and γ, as well as relations between these variables that must always be respected, but no concrete value is defined. The problem is therefore to find an admissible assignment for these variables, based on the constraints present in the inference rules of Trocq.

Univalent solution A solution to this problem always exists: it suffices to set all classes to $(4,4)$, which corresponds to a univalent parametricity translation. Let us explain why such a solution is always valid. First, the various possible constraints in the traversal can always be reduced to order constraints between the variables involved or equality to $(4,4)$. A constraint of the form $(\alpha, \beta) \in \mathcal{D}_\square$ can be reduced when the value of β is known, by distinguishing two cases: if β belongs to $\{0,1,2_\mathrm{a}\}^2$, then α is not constrained; otherwise, $\alpha = (4,4)$. The procedure setting all the classes to $(4,4)$ is therefore compatible with this constraint: if we set β to $(4,4)$, the constraint implied on α is compatible since we also set α to $(4,4)$. A constraint of the form $(\alpha, \beta) = \mathcal{D}_\star(\gamma)$ where $\star \in \{\to, \Pi\}$ can be reduced to two order constraints when the value of γ is known. If we set γ to $(4,4)$, then the dependency tables in Figure 9.2 yield $\alpha = \beta = (4,4)$, which is compatible since we also set these classes to $(4,4)$. Finally, order constraints between classes are never strict, so two classes α and β set at $(4,4)$ always satisfy an $\alpha \geq \beta$ constraint. Consequently, the solution that can be described as *univalent*, is always valid.

Search for a minimal solution However, the univalent solution is not satisfactory since it always requires the maximum amount of information from the user and at least one axiom whenever a universe or dependent product is present in the initial goal. Therefore, we look for another solution, with the opposite objective: the less information we ask for in general, the more goals we can process from a given user knowledge base, and the less likely we are to need an axiom. It seems natural to try to minimise all parametricity classes. However, it is not sure that such a solution can be built, or that it actually achieves the objective.

On the one hand, a minimisation order for the parametricity classes present in the constraints must be determined. However, adding constraints during traversal is not totally structural. Not all constraints on a node in the initial goal are determined when the node is traversed. For example, any occurrence of a bound variable may induce the addition of a constraint on the parametricity classes present in its type, even though this type has already been processed earlier, when the binder introducing the bound variable was traversed. It is therefore not obvious that minimising the variables in the order in which they are encountered is the best solution. Furthermore, in the absence of a more detailed study of the behaviour of constraints according to the structure of the goal, we cannot rule out the possibility that, for some goals, minimising one class of parametricity may impair the minimisation of another class of parametricity. In such a case, minimising one class before the other would increase the second one, implying the

use of an axiom that could have been avoided by changing the minimisation order of the classes. This case seems more likely to occur in the presence of constants in the initial goal, since the valid assignments of the parametricity classes in the type of a constant are specific to that constant and *a priori* do not respect any property common to all constants. The minimisation order therefore seems difficult to define in an analytical way.

On the other hand, still in the case of constants, minimisation may not be an optimal solution. The dependency tables in Figure 9.2 suggest that the classes required on the domain and codomain of a function type increase with the class required on that type. However, although this is intuitive in the case of functional types, without further study it cannot be said that this property, that could be described as *monotonicity*, generalises to all constants. In the case of a non-monotonic constant, it could be counter-productive to attempt to minimise all the classes. Consequently, even the action of trying to minimise all the parametricity classes present in the constraints could be an imperfect heuristic.

13.3.2 Solution chosen in Trocq

In the context of the implementation of Trocq, we propose an empirical solution whose optimality is not guaranteed in theory. In this solution, it is assumed that the entire computation is monotonic, so minimising the parametricity classes is a good heuristic. It is left to determine the minimisation order.

Complex constraints A key point about the minimisation order is that, when minimising a particular parametricity class, in order to select the minimal possible class, all the order constraints associated with that class must be known. This is because the constraints added during traversal of the goal are either order constraints or *complex* constraints, involving a dependency table in the same way as in Figure 9.2. When the construct in question is a type constructor, it lives in a universe equipped with a parametricity class that we will call an *output class*. The dependency table then has one row per parametricity class, in which we can read the various dependencies on the other parametricity classes linked to this construct. These dependencies are obtained by first determining the output parametricity class. Once we know these dependencies, we can replace the complex constraint with a list of order constraints from each of the classes present in the row of the table. For each complex constraint, we therefore know that the variable corresponding to the output class must be instantiated before the others, in order to *reduce* the complex constraint into one or more order constraints.

For example, the goal used to define the class inference problem in § 13.3.1 is a dependent product. The TrocqPi rule is used to determine at which types we should process the domain $\square^\alpha \to \square^\beta$ and the codomain $\Pi A : \square^\gamma.\, F\, A \to F\, A$, using a constraint $(\alpha, \beta) = \mathcal{D}_\Pi(\gamma)$. The goal traversal starts with an annotated type $\square^{(0,1)}$, *i. e.*, it sets the output class to $(0,1)$. We therefore have $\gamma = (0,1)$, and it is only with this information that we can find the right row in the table of Figure 9.2 allowing us to determine α and β.

Order constraints In an order constraint, a variable can be in a higher or lower position. To find the minimal class for a variable, we consider all the order constraints in which this variable is in a higher position, as these constraints give lower bounds for this variable. The minimal class is then the largest of the lower

bounds. In the case where two variables are linked by an order constraint $\alpha \geq \beta$, β is a lower bound for α. Consequently, to determine the minimal class for α, we must first know the ground class adopted for β. This order constraint therefore also constrains the minimisation order of both variables.

The global priority order is therefore determined using the following rules:

- $(\alpha, \beta) \in \mathcal{D}_{\square}$ reduces to an equality constraint on α or no constraint, depending on the value of β : it therefore forces variable β to be instantiated before α ;
- $\mathcal{D}_{\Pi}(\gamma) = (\alpha, \beta)$ or $\mathcal{D}_{\rightarrow}(\gamma) = (\alpha, \beta)$ forces γ to be instantiated before α and β ;
- $\alpha \geq \beta$ forces β to be instantiated before α ;
- $\mathcal{D}_{K}(\gamma, T)$ forces γ to be instantiated before the parametricity classes present in the annotated type T.

13.3.3 Implementation

The role of the inference algorithm is therefore to generate the minimisation order from the various constraints that appear during traversal, and then to select the minimal possible class for each variable, with each assignment reducing complex constraints into new order constraints on other variables yet unassigned. This algorithm can be implemented in several ways.

Finite domain constraint solving A first idea is to note that the problem of parametricity class inference closely resembles a problem of finite domain constraint solving. Such a problem can be elegantly solved in the style of *Constraint Logic Programming* [82], idiomatic in Prolog-based languages such as Elpi. Indeed, each unknown parametricity class in the initial goal can be represented as a variable with an initial domain ranging from $(0, 0)$ to $(4, 4)$, provided with a partial order. The various order constraints added on the variables reduce their domain, leaving only valid solutions. Complex constraints are put asleep until their output class is known, and when it is, they are automatically reduced and replaced by new order constraints. Each added constraint is also a constraint on the minimisation order. Thus, at the end of the traversal, the minimisation order emerges naturally and it suffices to calculate the largest lower bound of each variable following this order to obtain the desired solution.

[82]: Jaffar *et al.* (1987), "Constraint Logic Programming"

As we are dealing with parametricity classes and not integers, the domain of variables is not classic, therefore we need to implement an *ad hoc* constraint solver. *Constraint Handling Rules* (CHR), available in Elpi and presented earlier in § 4.1.1, are one of the best-known methods to express complex algorithms involving the generation and management of constraints with rules. The CHR language is well suited to the design of prototype constraint solvers, because the central ingredients of these solvers, constraint propagation and consistency checking,[14] can be implemented as rules. The constraint solver combines these rules with a search procedure that tests the remaining values in the domains after propagation. In our case, rules are used to simplify complex constraints and reduce them to order constraints on parametricity classes.

14: The consistency property is the compatibility at all times between the domains of variables and the various constraints on these variables.

Direct style However, the elegance of such a solution comes at the cost of trackability of control flow in the reduction process. Indeed, in constraint solvers, when constraints are declared, after an initial propagation phase, they actually remain in the constraint store in an asleep state, watching the variables they bound together, to be awaken every time one of their domains is updated. This behaviour, necessary to ensure consistency at any time, makes control flow very complex as it is difficult to know when propagation will happen just by reading the code, thus making debugging phases harder.

Instead, we present an inference algorithm in a more pragmatic style, first accumulating constraints in a *global constraint graph*, then reducing it and instantiating the variables afterwards. In this constraint graph, nodes are parametricity classes — variable or ground, and different kinds of edges exist, one for each kind of constraint possibly added during traversal of the goal. An example of constraint graph is available in Figure 13.2. For instance, the edges from X_1 to X_2 and X_3 represent the constraint $(X_2, X_3) = \mathcal{D}_\Pi(X_1)$, and the edge from X_6 to the constant class $(3, 2_\mathrm{b})$ represents the constraint $(3, 2_\mathrm{b}) \geq X_6$.

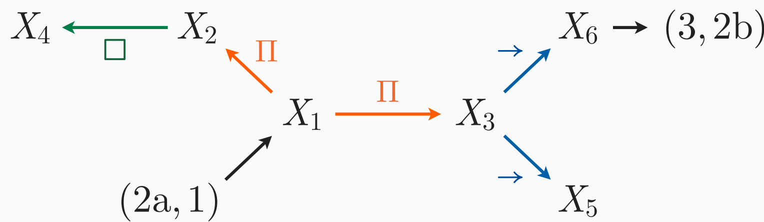


Figure 13.2: Example of constraint graph

A special aspect of the constraint graph is that the direction of the edges is consistent with the minimisation order of parametricity classes. Thus, in the graph above, the edge from X_1 to X_2 means that there is a constraint on a dependent product but also that variable X_1 is the output class, so it must be instantiated before X_2. The same applies to other kinds of constraints: X_3 must be instantiated before X_5 and X_6, X_2 before X_4, *etc.* This means that the constraint graph is also a *priority graph*. The final instantiation order can then be obtained by performing a *topological sort* on this graph: input nodes are recursively removed by adding them to a list, to obtain an admissible minimisation order, all this while ignoring nodes that are not variables. For example, in the graph above, an admissible order is $[X_1, X_2, X_4, X_3, X_5, X_6]$. If we respect the minimisation order, when a variable is instantiated, all the edges pointing to this variable, *i. e.*, all the lower nodes, must form a set of constant classes. Each time a variable is instantiated, the corresponding node is removed from the graph. Then, the complex constraints for which the freshly instantiated variable is the output class can now be reduced to order constraints and added back to the graph. In this way, the invariant is maintained and the next variable is ready to be instantiated. For example, when X_1 is instantiated with the value $(2_\mathrm{a}, 1)$, the node disappears and we can reduce the constraint $(X_2, X_3) = \mathcal{D}_\Pi(X_1)$ into two order constraints $X_2 \geq (2_\mathrm{a}, 4)$ and $X_3 \geq (2_\mathrm{a}, 1)$ that are added back to the graph. The next variable in the minimisation order is X_2, and the only constraint pointing to the associated node in the graph is this new order constraint. So there are no more complex constraints and instantiation can proceed. The process continues in this way until all the nodes have been removed from the graph.

13.3.4 Weakening and subtyping

The addition of constraints on a parametricity class ensures that the class assigned to it after translation matches the needs of the various occurrences of this variable, *i. e.*, that the associated parametricity witness has sufficient information in all its occurrences in the global witness. However, this does not mean that all instances need the same parametricity witness. It is very likely that at least one occurrence of the variable requires less information than is contained in the witness. In such a case, to guarantee the well-typedness of the proof term, a weakening function must be inserted in front of the witness. The nature of this function can only be determined when we know the source class and the target class of the weakening.

For example, let us take the goal processed in § 13.3.1:

$$\Pi F : \square^{\alpha} \to \square^{\beta}. \Pi A : \square^{\gamma}. F\ A \to F\ A$$

In the trace of the traversal, we can see that the subterm $F\ A$ is processed twice, at type $\square^{(1,0)}$ and then at type $\square^{(0,1)}$. During the traversal, variables F and A are given associated variables F' and A'. In the global parametricity witness, at the position of the first occurrence of $F\ A$, the witness relating it to $F'\ A'$ must have type $\mathrm{Param}^{(1,0)}\ (F\ A)\ (F'\ A')$, and at its second occurrence, a proof of type $\mathrm{Param}^{(0,1)}\ (F\ A)\ (F'\ A')$ is expected. Both occurrences therefore involve witnesses of different types, both subtypes of the final type that will be retained after the class inference phase. However, this final type is not known during the traversal, yet it is necessary to add a weakening function in front of the witness F_R in the proof term at the moment the occurrences of $F\ A$ are traversed.

In order to solve this problem, in the implementation of Trocq, weakenings are represented syntactically with an identity with phantom arguments associated with the parametricity classes present in the constraint graph:

```
Definition weaken (m1 n1 m2 n2 : map_class) {A : Type} (a : A) := a.
```

This allows keeping a well-formed and well-typed term, while adding information that is visible when the term is traversed. Once the variables have been assigned, a completion procedure is run, replacing these placeholders with true weakening functions, that can be generated at this point, since the placeholder contains the two necessary ground parametricity classes.

Note that weakening involves a suspension in the cases of the abstraction and the application. The purpose of this suspension is to handle the case of a β-redex, *i. e.*, an application with an abstraction at its head. Suspension is implemented by an Elpi data type that can contain the body of an abstraction in order to delay the creation of the weakening function until the arguments supplied to this abstraction have been read. Once again, we exploit the HOAS representation of Coq terms with Elpi meta-functions.

13.4 Universe polymorphism

Throughout this document, both in parts discussing theory and parts related to implementation, we presented λ-terms in calculi equipped with a hierarchy of universes where universes are indexed by an integer representing their level. The type system then allows giving to each universe $\square_i$ a type, its successor $\square_{i+1}$, or any universe with a strictly higher level thanks to the cumulativity rule. However,

the various parametricity lemmas we described are *universe-polymorphic*, *i. e.*, they work with types from any universes as long as these universes respect the various constraints imposed by the definition of the lemma. For example, the rule TrocqPi uses three universe levels i, j, and k, that can take any value as long as the constraint $k = \max(i, j)$ is respected.

When using parametricity lemmas, bound universes must therefore be instantiated to form valid terms. However, the important property for typing is just *existence* of an admissible value for each bound universe. Therefore, when we are sure that there is always a solution to the problem of universe constraints, we can afford leaving universes implicit. For example, a parametricity translation such as those presented in this thesis is a structural translation that does not introduce additional universes or additional universe constraints with respect to the context of the initial goal, and does not aim to reason about universes. As there is no risk of introducing inconsistencies, we can afford ignoring universes in our theoretical presentations.

At the time of implementation, universes can also be left implicit because the Coq kernel can infer them during typechecking. However, in the context of universe polymorphism, this inference is less powerful because it must also infer bound universes.[15] It then becomes necessary to make explicit some universes present in the terms. However, universe polymorphism is still an experimental feature in Coq-Elpi. In addition, some universes in Trocq are *algebraic*, *i. e.*, they are expressed using other universes and arithmetic operations, and the current implementation of Coq is limited regarding this feature. This section therefore details the issues that arise when implementing Trocq, about typical ambiguity and algebraic universes.

15: In the rest of this section, we consider that we are in this context.

13.4.1 Clearing typical ambiguity

When a universe-polymorphic constant is defined in Coq, the universe variables present in the term become bound universes. For example, in the following definition, `A` and `B` are types each living in a fresh universe (say u_0 and u_1 respectively), but these universes are left implicit thanks to *typical ambiguity*, a feature in Coq that allows automated inference of universe levels:

```
Definition twice (A : Type) (B : Type) (f : A → A → B) : A → B :=
  fun a ⇒ f a a.
```

When the term is given to Coq to be stored as a constant and named `twice`, levels u_0 and u_1 are replaced with bound universes, i and j respectively. The `twice` constant is then an incomplete term awaiting a *universe instance*, *i. e.*, an array of universe levels that gives a value to each bound universe. Once again, thanks to typical ambiguity, the universe instance can be omitted when using the constant and Coq takes care of the inference, allowing users to benefit from the advantages of universe polymorphism without making the terms syntactically heavier.

However, this inference is flawed because it does not minimise the number of bound universes. Indeed, there are cases in which a universe-polymorphic constant with at least two bound universes can be instantiated with the same universe several times within the same instance. Yet, in such a case, Coq will allocate a different universe variable for each bound universe in the instance. For example, if the constant `twice` defined above is used in another term, it will appear

instantiated by Coq, *i. e.*, `twice@{u3 u4}`, even though it is possible to instantiate it twice with `u3`. If the term containing such a constant is then defined in Coq, these universes become bound. As a result, the universe instance of the global term contains more bound universes than necessary, leading in some cases to subsequent ill-typed terms.

To guarantee the well-typedness of terms generated by a meta-program, we must determine whether such cases are possible in order to know whether typical ambiguity can be used. In the case of Trocq's implementation, many universes had to be made explicit. Indeed, by carefully defining parametricity lemmas, it is theoretically possible to guarantee the invariant that a parametricity witness requires as many universes as the initial goal. For this purpose, parametricity lemmas must be defined by imposing a maximum size constraint on their universe instance, which disturbs universe inference in Coq. Most universes then have to be annotated manually to have maximum control over the term that is actually defined. To achieve this, it was necessary to add new functions to Coq-Elpi to allow handling universe-polymorphic terms, universe instances, *etc.*

For example, the parametricity lemma used to build a parametricity witness for a dependent product, presented in § 13.1.1, is actually defined as follows in Coq:

```
Definition Param^γ_Π@{i j k | i ⩽ k, j ⩽ k}
  (A A' : Type@{i}) (A_R : Param^α@{i} A A')
  (B : A → Type@{j}) (B' : A' → Type@{j})
  (B_R : forall a a' a_R, Param^β@{j} (B a) (B' a')) :
    Param^γ@{k} (forall (a : A), B a) (forall (a' : A'), B' a').
```

Making universes explicit in the type of the definition in fact amounts to disabling typical ambiguity, as Coq is no longer allowed to infer more related universes or constraints than specified in the header of this definition, which leads to annotating the term by hand. Indeed, in the context of universe polymorphism, the proof assistant can add universe constraints that we have not anticipated or that we do not want. These inferred constraints are logical and necessary for the term to be well typed, but they can result from poor manual annotation. If the universe instance is blocked by making the definition header explicit as above, then these additional silent constraints become typing errors. By iterating on the errors supplied, we can then design the definition corresponding to the minimal annotations desired. The equivalent of this header in Coq-Elpi is the following:

```
@udecl! [I, J, K] ff [le I K, le J K] ff =>
  coq.env.add-const %...
```

Value `ff` represents boolean *false*, indicating that the lists of universes and constraints cannot be extended by Coq. Changing one of these booleans to *true* is equivalent to adding a `+` after one of the two lists in the Coq definition, and gives the proof assistant the freedom to add values. A constant `K` can then be instantiated with `pglobal K UI`, where `UI` is the universe instance obtained from a list of universe variables:

```
coq.univ-instance UI [I0, J0, K0]
```

13.4.2 Algebraic universes and bound universes

Actually, the size of universe instances in the parametricity lemmas could be further reduced. Creating a new fresh universe for a dependent product and forcing

it to be greater than the domain and codomain's universes, as done in the parametricity lemma presented in the last subsection, can be avoided by using *algebraic* universes, *i. e.*, universes created from other universes with two operations: maximum and successor. Indeed, the typing rule for the dependent product involves only two universes:

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

Thus, the parametricity lemma theoretically requires only two bound universes, and the lemma for universes requires only one, since a universe is always related to itself. As a result, it is possible to maintain the invariant that the associated goal and the parametricity witness use as many universes as the initial goal.

However, the current implementation of universe polymorphism in Coq does not allow this kind of definition. This is because the parametricity lemma on dependent products builds a parametricity witness in the form of a record, *i. e.*, an inductive type. Such a value is necessarily obtained by the corresponding record constructor. However, for a technical reason in the current implementation of the universe constraint graph verification algorithm in Coq, it is impossible to instantiate a constant with an algebraic universe. It is therefore necessary to use a fresh universe and additional constraints.

So, for each dependent product or universe appearing in the input term, the implemented parametricity translation creates a fresh universe. Furthermore, without a specific memory mechanism, encountering the same universe twice creates two different fresh universes. All this makes it difficult to track universes used in the traversal of the goal.

Finally, this problem prevents the implementation of a forward chaining proof transfer feature within Trocq. Indeed, rather than translating an initial goal G to be proved within Coq, it could be interesting, by flipping all the parametricity witnesses provided by the user in the knowledge base, to translate a lemma p' into a lemma p usable in a proof in the context selected by the user for their formalisation. However, in such a case, at the time of definition, the header would contain many useless bound universes, making it difficult to instantiate the term in an optimal way afterwards. We could then obtain ill-typed proofs if we let Coq carry out inference of the universe instance, and manual annotation would require significant and artificial work since these bound universes have no relevant *raison d'être*.

In the context of the use of universe polymorphism in Coq-Elpi, various predicates can be developed, such as a `coq.univ.super` predicate to obtain the successor of a universe, or `coq.univ.max` to obtain the maximum of two universes, thus making it possible to forge arbitrary algebraic universes. However, despite the possibility of making these low-level details accessible from the meta-language, when the term is translated into Coq, the algebraic universe will inevitably become a fresh universe variable accompanied by universe constraints.

Conclusion and perspectives

Contributions

In this document, we have presented both prototypes of proof transfer plugins for the Coq proof assistant developed throughout this thesis. The general context of this work is the search for solutions to allow a user to employ several formalisations of the same mathematical concept in their proofs in a transparent way, while keeping interoperability between all the proofs performed on a given theory, regardless of the representation chosen in each proof.

The first prototype, Trakt, improves proof automation for statements from the SMT family by reformulating these statements and expressing them in a canonical form adapted to the input format of the proof automation tools available in Coq. The project adds support for the theory of congruence and uninterpreted functions to the existing automation tools, as well as more flexible logic processing, allowing adaptation of the goal to the needs of various automation tactics. The `trakt` tactic has been successfully integrated into the SMTCoq plugin *via* the Sniper project, making several goal pre-processing tools work together.

In Trakt, different representations of a given mathematical object are related by isomorphisms, or partial embeddings in the case of subtyping, and the tool focuses on goals of the SMT family, which are the target of the automated provers that we wish to execute after pre-processing. This preprocessing bridging the gap between automated and interactive theorem proving, is an instance of the more general problem of proof transfer, target of the second prototype, Trocq. The main ingredient of this second contribution is a new modular parametricity framework able to accomodate a more general class of relations than the previous approaches of raw and univalent parametricity. The modularity of Trocq lies in a hierarchy of parametricity witnesses exploited to perform proof transfer while avoiding as much as possible the use of axioms when they are not necessary.

Perspectives

The social objective of formal proof is to establish greater trust between humans thanks to greater trust in the results of researchers, engineers, logicians, *etc.*, *i. e.*, certified proofs and computer programs. The various current implementations of proof assistants all derive from different paths in the search for the best logical framework to carry out these formal proofs, and within each of these tools, there exists a wide spectrum of formalisation techniques. This Cambrian explosion makes it possible to explore a vast space of possibilities, but isolates the various formalisation efforts from each other.

Any good practices brought from the domains of software development and programming language theory are therefore welcome for users of proof assistants who, *de facto*, by choosing this kind of software, accept the compromise of a high level of confidence at the price of more — often way too — manual proofs. Ideally, a user carrying out formalisation work would like to be able to use the

data structures they consider to be the most practical, without having to manage equivalences manually for their development to conform to the encodings commonly used by the community of the proof assistant they chose. Conversely, if a formalisation has already been carried out using an encoding of a mathematical concept, the user wants to be able to use this work with another encoding of the same concept, given that they are equivalent. Proof transfer mechanisms are an excellent intermediary tool in this kind of situation, and help factorise proof efforts.

Nevertheless, there is still a step to be taken to turn the prototypes developed during this thesis into real proof assistance tools. Indeed, as proof assistants are research objects, the development of their tooling is being done at the same time as the development of their logical formalism itself and of meta-programming techniques. Although some of the foundations of these tools are already stable, such as support for the Calculus of Constructions, or the use of logic programming to perform syntactic translations, certain subtleties, mainly related to universes, remain unstable and are delaying their maturity.

From a more practical point of view, these prototypes could be made more usable by pushing further the commands used to add information to the database. In the case of Trocq, we could imagine a command that generates all possible parametricity witnesses relating an inductive type to itself. This kind of proof seems to be always possible in theory, and would allow transfer of user data types within an arbitrarily complex goal, for free. Management of the impredicative universe $\mathbb{P}$ would bring the tool closer to the standard version of Coq rather than relying on the HoTT library. Finally, we can imagine, for both Trakt and Trocq, building proof libraries to be added to the database as soon as the plugin is imported, so that the user can effortlessly start performing proof transfer when it concerns data types that are widespread in formalisations. A mathematician might enjoy Trocq to be equipped with a library of parametricity witnesses already proven for the data types in the MathComp library, so as not to have to carry out these proofs themselves, and to be able to easily turn, *e. g.*, a matrix into a list of lists.

The completion of these proof transfer tools would make it simpler to reason modulo equivalence within the proof assistant. Bringing the construction of formal proofs closer to the intuitive reasoning that can be done on paper attracts new users, moving a little closer to a world without software faults, with all the implied societal benefits.

Bibliography

Here are the references in citation order.

[1] Gottlob Frege. "Begriffsschrift : Eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens." In: (1882) (cited on page 3).

[2] Giuseppe Peano. "Arithmetices principia : Nova methodo exposita." In: (1889) (cited on page 3).

[3] Georg Cantor. "Grundlagen einer allgemeinen Mannigfaltigkeitslehre. Ein mathematisch-philosophischer Versuch in der Lehre des Unendlichen." In: (1883) (cited on page 3).

[4] Alain Colmerauer *et al.* "Un système de communication homme-machine en français." In: *Rapport préliminaire, Groupe de Recherche en Intelligence Artificielle* (1973) (cited on pages 3, 34).

[5] Leslie Lamport. "The Temporal Logic of Actions." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.3 (1994), pp. 872–923 (cited on page 3).

[6] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 — Where Programs Meet Provers." In: *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer. 2013, pp. 125–128 (cited on page 3).

[7] Niki Vazou. *Liquid Haskell: Haskell as a Theorem Prover*. University of California, San Diego, 2016 (cited on page 3).

[8] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002 (cited on pages 4, 29).

[9] Ulf Norell. "Dependently Typed Programming in Agda." In: *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. Ed. by Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra. Vol. 5832. Lecture Notes in Computer Science. Springer, 2008, pp. 230–266. doi: 10.1007/978-3-642-04652-0_5 (cited on page 4).

[10] Leonardo De Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language." In: *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*. Ed. by André Platzer and Geoff Sutcliffe. Vol. 12699. Lecture Notes in Computer Science. Springer, 2021, pp. 625–635. doi: 10.1007/978-3-030-79876-5_37 (cited on page 4).

[11] The Coq Development Team. *The Coq Proof Assistant*. Version 8.16. Sept. 2022. doi: 10.5281/zenodo.7313584 (cited on pages 4, 7).

[12] Valentin Blot *et al.* "Compositional pre-processing for automated reasoning in dependent type theory." In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2023, pp. 63–77 (cited on pages 5, 42, 51, 63, 66).

[13] John C. Reynolds. "Types, Abstraction and Parametric Polymorphism." In: *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. Ed. by R. E. A. Mason. North-Holland/IFIP, 1983, pp. 513–523 (cited on pages 5, 33, 73).

[14] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. "The marriage of univalence and parametricity." In: *Journal of the ACM (JACM)* 68.1 (2021), pp. 1–44 (cited on pages 5, 71, 75, 80, 92, 98).

[15] Cyril Cohen, Enzo Crance, and Assia Mahboubi. "Trocq: Proof Transfer for Free, With or Without Univalence." In: *European Symposium on Programming*. 2024 (cited on pages 5, 72).

[16] Enrico Tassi. "Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect)." In: (2018) (cited on pages 5, 8, 34).

[17] Frédéric Besson. "Fast Reflexive Arithmetic Tactics the Linear Case and Beyond." In: *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*. Ed. by Thorsten Altenkirch and Conor McBride. Vol. 4502. Lecture Notes in Computer Science. Springer, 2006, pp. 48–62. doi: 10.1007/978-3-540-74464-1_4 (cited on pages 7, 29, 43).

[18] Alonzo Church. “A set of postulates for the foundation of logic.” In: *Annals of mathematics* (1933), pp. 839–864 (cited on page 9).

[19] Alan M Turing. “Computability and λ-definability.” In: *The Journal of Symbolic Logic* 2.4 (1937), pp. 153–163 (cited on page 9).

[20] Alonzo Church. “A formulation of the simple theory of types.” In: *The journal of symbolic logic* 5.2 (1940), pp. 56–68 (cited on page 10).

[21] Henk P Barendregt. “Introduction to generalized type systems.” In: (1991) (cited on page 11).

[22] Stefano Berardi. “Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt’s cube.” In: *Technical report, Carnegie-Mellon University (USA) and Università di Torino (Italy)* (1988) (cited on page 12).

[23] Jan Terlouw. “Een nadere bewijstheoretische analyse van GSTT’s.” In: *Manuscript* (1989) (cited on page 12).

[24] Thierry Coquand and Gérard Huet. “The calculus of constructions.” PhD thesis. INRIA, 1986 (cited on page 13).

[25] Jean-Yves Girard. “Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur.” In: (1972) (cited on page 13).

[26] Gottlob Frege. *Grundgesetze der Arithmetik*. 1903 (cited on page 13).

[27] Matthieu Sozeau and Nicolas Tabareau. “Universe polymorphism in Coq.” In: *International Conference on Interactive Theorem Proving*. Springer. 2014, pp. 499–514 (cited on page 16).

[28] Christine Paulin-Mohring. “Définitions Inductives en Théorie des Types.” PhD thesis. Université Claude Bernard-Lyon I, 1996 (cited on page 16).

[29] Philip Wadler. “Monads for functional programming.” In: *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1*. Springer. 1995, pp. 24–52 (cited on page 20).

[30] Roger 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 22).

[31] Robin Milner. “A theory of type polymorphism in programming.” In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375 (cited on page 22).

[32] Cordelia V Hall *et al.* “Type classes in Haskell.” In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18.2 (1996), pp. 109–138 (cited on page 25).

[33] Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. “Type classes as objects and implicits.” In: *ACM Sigplan Notices* 45.10 (2010), pp. 341–360 (cited on page 25).

[34] Matthieu Sozeau and Nicolas Oury. “First-class type classes.” In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2008, pp. 278–293 (cited on page 25).

[35] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, Jan. 2021 (cited on pages 26, 45).

[36] 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*. 167. 2020, pp. 34–1 (cited on page 26).

[37] Assia Mahboubi and Enrico Tassi. “Canonical structures for the working Coq user.” In: *International Conference on Interactive Theorem Proving*. Springer. 2013, pp. 19–34 (cited on page 26).

[38] David Delahaye. “A tactic language for the system Coq.” In: *Logic for Programming and Automated Reasoning: 7th International Conference, LPAR 2000 Reunion Island, France, November 6–10, 2000 Proceedings 7*. Springer. 2000, pp. 85–95 (cited on page 26).

[39] Sascha Böhme and Tobias Nipkow. “Sledgehammer: judgement day.” In: *Automated Reasoning: 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings 5*. Springer. 2010, pp. 107–121 (cited on page 29).

[40] Łukasz Czajka and Cezary Kaliszyk. “Hammer for Coq: Automation for dependent type theory.” In: *Journal of automated reasoning* 61 (2018), pp. 423–453 (cited on pages 29, 61).

[41] Burak Ekici *et al.* "SMTCoq: A Plug-In for Integrating SMT Solvers into Coq." In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 126–133. doi: 10.1007/978-3-319-63390-9_7 (cited on pages 29, 43).

[42] Clark Barrett, Aaron Stump, Cesare Tinelli, *et al.* "The SMT-LIB Standard: Version 2.0." In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*. Vol. 13. 2010, p. 14 (cited on page 29).

[43] Matthieu Sozeau. "A New Look at Generalized Rewriting in Type Theory." In: *J. Formaliz. Reason.* 2.1 (2009), pp. 41–62. doi: 10.6092/issn.1972-5787/1574 (cited on pages 32, 98).

[44] Gilles Barthe, Venanzio Capretta, and Olivier Pons. "Setoids in type theory." In: *J. Funct. Program.* 13.2 (2003), pp. 261–293. doi: 10.1017/S0956796802004501 (cited on page 32).

[45] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. "Refinements for Free!" In: *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*. Ed. by Georges Gonthier and Michael Norrish. Vol. 8307. Lecture Notes in Computer Science. Springer, 2013, pp. 147–162. doi: 10.1007/978-3-319-03545-1_10 (cited on pages 32, 98).

[46] Matthieu Sozeau *et al.* "The MetaCoq Project." In: *J. Autom. Reason.* 64.5 (2020), pp. 947–999. doi: 10.1007/s10817-019-09540-0 (cited on pages 34, 98).

[47] Pierre-Marie Pédrot. "Ltac2: tactical warfare." In: *The Fifth International Workshop on Coq for Programming Languages, CoqPL*. 2019, pp. 13–19 (cited on page 34).

[48] Cvetan Dunchev *et al.* "ELPI: Fast, Embeddable, λProlog Interpreter." In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer. 2015, pp. 460–468 (cited on page 34).

[49] Dale Miller and Gopalan Nadathur. *A logic programming approach to manipulating formulas and programs*. University of Pennsylvania. Moore School of Electrical Engineering . . ., 1987 (cited on page 35).

[50] Frank Pfenning and Conal Elliott. "Higher-Order Abstract Syntax." In: *ACM sigplan notices* 23.7 (1988), pp. 199–208 (cited on page 35).

[51] Thom Frühwirth. "Constraint Handling Rules." In: *French School on Theoretical Computer Science*. Springer, 1994, pp. 90–107 (cited on pages 36, 115).

[52] Nicolaas Govert De Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem." In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5. Elsevier. 1972, pp. 381–392 (cited on page 37).

[53] Frédéric Besson. "ppsimpl: a reflexive Coq tactic for canonising goals." In: *Coq Workshop on Programming Languages*. https://popl17.sigplan.org/details/main/3/ppsimpl-a-reflexive-Coq-tactic-for-canonising-goals. 2017 (cited on page 46).

[54] Kazuhiko Sakaguchi. *Micromega tactics for Mathematical Components*. Version 1.12. 2019–2022 (cited on page 49).

[55] Frédéric Besson. "Itauto: An Extensible Intuitionistic SAT Solver." In: *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*. Ed. by Liron Cohen and Cezary Kaliszyk. Vol. 193. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 9:1–9:18. doi: 10.4230/LIPIcs.ITP.2021.9 (cited on pages 49, 62).

[56] Andrew W. Appel. "Verified Software Toolchain - (Invited Talk)." In: *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Ed. by Gilles Barthe. Vol. 6602. Lecture Notes in Computer Science. Springer, 2011, pp. 1–17. doi: 10.1007/978-3-642-19718-5_1 (cited on page 63).

[57] Andrew W. Appel *et al. Verifiable C*. 2022 (cited on page 63).

[58] John C. Mitchell. "Representation Independence and Data Abstraction." In: *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. ACM Press, 1986, pp. 263–276. doi: 10.1145/512644.512669 (cited on page 71).

[59] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. "The next 700 syntactical models of type theory." In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. Ed. by Yves Bertot and Viktor Vafeiadis. ACM, 2017, pp. 182–194. doi: 10.1145/3018610.3018620 (cited on pages 71, 79).

[60] Philip Wadler. "Theorems for Free!" In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*. Ed. by Joseph E. Stoy. ACM, 1989, pp. 347–359. doi: 10.1145/99370.99404 (cited on page 74).

[61] Jean-Philippe Bernardy and Marc Lasson. "Realizability and Parametricity in Pure Type Systems." In: *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Ed. by Martin Hofmann. Vol. 6604. Lecture Notes in Computer Science. Springer, 2011, pp. 108–122. doi: 10.1007/978-3-642-19805-2_8 (cited on page 74).

[62] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. "Proofs for free - Parametricity for dependent types." In: *J. Funct. Program.* 22.2 (2012), pp. 107–152. doi: 10.1017/S0956796812000056 (cited on page 74).

[63] Chantal Keller and Marc Lasson. "Parametricity in an Impredicative Sort." In: *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*. Ed. by Patrick Cégielski and Arnaud Durand. Vol. 16. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012, pp. 381–395. doi: 10.4230/LIPIcs.CSL.2012.381 (cited on page 74).

[64] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013 (cited on pages 75, 77, 83).

[65] Andrej Bauer *et al.* "The HoTT library: a formalization of homotopy type theory in Coq." In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. Ed. by Yves Bertot and Viktor Vafeiadis. ACM, 2017, pp. 164–172. doi: 10.1145/3018610.3018615 (cited on pages 78, 82).

[66] Talia Ringer *et al.* "Proof repair across type equivalences." In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 112–127. doi: 10.1145/3453483.3454033 (cited on page 79).

[67] David Aspinall and Adriana B. Compagnoni. "Subtyping dependent types." In: *Theor. Comput. Sci.* 266.1-2 (2001), pp. 273–309. doi: 10.1016/S0304-3975(00)00175-4 (cited on page 93).

[68] Talia Ringer *et al.* "Proof repair across type equivalences." In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 112–127. doi: 10.1145/3453483.3454033 (cited on page 98).

[69] Gilles Barthe and Olivier Pons. "Type Isomorphisms and Proof Reuse in Dependent Type Theory." In: *Foundations of Software Science and Computation Structures*. Ed. by Furio Honsell and Marino Miculan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 57–71 (cited on page 98).

[70] Nicolas Magaud. "Changing Data Representation within the Coq System." In: *TPHOLs'2003*. Vol. 2758. © Springer-Verlag. LNCS, Springer-Verlag, 2003 (cited on page 98).

[71] Neelakantan R. Krishnaswami and Derek Dreyer. "Internalizing Relational Parametricity in the Extensional Calculus of Constructions." In: *Computer Science Logic 2013 (CSL 2013)*. Ed. by Simona Ronchi Della Rocca. Vol. 23. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 432–451. doi: 10.4230/LIPIcs.CSL.2013.432 (cited on page 98).

[72] Maxime Dénès, Anders Mörtberg, and Vincent Siles. "A Refinement-Based Approach to Computational Algebra in Coq." In: *Interactive Theorem Proving*. Ed. by Lennart Beringer and Amy Felty. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 83–98 (cited on page 98).

[73] Peter Lammich. "Automatic Data Refinement." In: *Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 84–99 (cited on page 98).

[74] Florian Haftmann *et al.* "Data Refinement in Isabelle/HOL." In: *Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 100–115 (cited on page 98).

[75] Brian Huffman and Ondřej Kunčar. "Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL." In: *Certified Programs and Proofs*. Ed. by Georges Gonthier and Michael Norrish. Cham: Springer International Publishing, 2013, pp. 131–146 (cited on page 98).

[76] Peter Lammich and Andreas Lochbihler. "Automatic Refinement to Efficient Data Structures: A Comparison of Two Approaches." In: *J. Autom. Reason.* 63.1 (2019), pp. 53–94. doi: 10.1007/s10817-018-9461-9 (cited on page 98).

[77] Théo Zimmermann and Hugo Herbelin. "Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant." In: *Conference on Intelligent Computer Mathematics*. Washington, D.C., United States, 2015 (cited on page 98).

[78] Carlo Angiuli *et al.* "Internalizing representation independence with univalence." In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–30. doi: 10.1145/3434293 (cited on page 98).

[79] Abhishek Anand and Greg Morrisett. *Revisiting Parametricity: Inductives and Uniformity of Propositions*. 2017 (cited on page 98).

[80] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. "Equivalences for free: univalent parametricity for effective transport." In: *Proceedings of the ACM on Programming Languages* 2.ICFP (2018), pp. 1–29 (cited on page 98).

[81] Xavier Allamigeon, Quentin Canu, and Pierre-Yves Strub. "A Formal Disproof of Hirsch Conjecture." In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*. Ed. by Robbert Krebbers *et al.* ACM, 2023, pp. 17–29. doi: 10.1145/3573105.3575678 (cited on page 99).

[82] Joxan Jaffar and Jean-Louis Lassez. "Constraint Logic Programming." In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1987, pp. 111–119 (cited on page 120).

COLLEGE DOCTORAL PAYS DE LA LOIRE | ED MATHEMATIQUES STIC

Nantes Université

Titre : Méta-programmation pour le transfert de preuve en théorie des types dépendants

Mot clés : preuve formelle, automatisation des preuves, méta-programmation

Résumé : En mathématiques comme en informatique, il est d'usage de faire appel à des outils numériques de vérification pour augmenter la confiance dans les preuves et les logiciels. La pratique la plus commune est le test, mais elle est limitée. Les *assistants de preuve interactifs* sont des outils permettant d'effectuer des preuves avec une grande confiance, laissant l'humain trouver les idées des preuves tout en vérifiant méticuleusement que toutes les étapes de la preuve sont valides. Cette thèse s'inscrit dans une lignée de travaux visant à automatiser les preuves, avec l'objectif final de répandre l'usage des assistants de preuve à la place du test logiciel, partout où cela est possible et pertinent. On s'intéresse ici au partage de théorie formelle entre plusieurs représentations différentes d'un même concept mathématique, ou plusieurs implémentations d'une même spécification. Sur le plan théorique, cette étude s'appuie sur l'analyse de traductions de paramétricité pour le Calcul des Constructions, et en propose une généralisation. Ces résultats s'incarnent dans la conception de deux outils de transfert de preuve, TRAKT et TROCQ, dont on discute ici l'implémentation, à l'aide du méta-langage COQ-ELPI.

Title: Meta-Programming for Proof Transfer in Dependent Type Theory

Keywords: formal proof, proof automation, meta-programming

Abstract: In both mathematics and computer science, it is common practice to use digital verification tools to increase confidence in proofs and software. The most common practice is testing, but it is limited. *Interactive proof assistants* are tools made to perform proofs with high confidence, letting humans come up with proof ideas while meticulously checking that all proof steps are valid. This thesis is part of a line of work aimed at automating proofs, with the ultimate goal of spreading the use of proof assistants in place of software testing, wherever possible and relevant. Here, we are interested in sharing of formal theory between several different representations of the same mathematical concept, or several implementations of the same specification. From a theoretical point of view, this study is based on the analysis of parametricity translations for the Calculus of Constructions, and proposes a generalisation of them. These results are made concrete in the design of two proof transfer tools, TRAKT and TROCQ, whose implementation is discussed here, using the COQ-ELPI meta-language.