Generic Proof Tools and Finite Group Theory

François Garillot

▶ To cite this version:

François Garillot. Generic Proof Tools and Finite Group Theory. Logic in Computer Science [cs.LO]. Ecole Polytechnique X, 2011. English. NNT : . pastel-00649586

HAL Id: pastel-00649586
https://pastel.hal.science/pastel-00649586v1

Submitted on 12 Jan 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

FRANÇOIS GARILLOT

GENERIC PROOF TOOLS AND FINITE GROUP THEORY

ÉCOLE POLYTECHNIQUE

. . . THE DESIGNER OF A NEW SYSTEM MUST NOT ONLY BE THE IMPLEMENTOR AND THE FIRST LARGE-SCALE USER; THE DESIGNER SHOULD ALSO WRITE THE FIRST USER MANUAL. . . IF I HAD NOT PARTICIPATED FULLY IN ALL THESE ACTIVITIES, LITERALLY HUNDREDS OF IMPROVEMENTS WOULD NEVER HAVE BEEN MADE, BECAUSE I WOULD NEVER HAVE THOUGHT OF THEM OR PERCEIVED WHY THEY WERE IMPORTANT.

DONALD E. KNUTH

Copyright (c) 2011 François Garillot

PUBLISHED BY ÉCOLE POLYTECHNIQUE

ÉCOLE DOCTORALE DE L'ÉCOLE POLYTECHNIQUE

First printing, November 2011

Contents

1 *Canonical Structures* 17

2 *Implementation* 83

3 *Subfunctors of the identity* 117

Conclusion 139

Bibliography 143

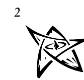
Colophon

This thesis was prepared in LaTeX and compiled with pdfTeX. It uses the `tufte-book` class from the `tufte-latex` project, a class made to replicate some of the designs of Pr. Edward R. Tufte. Crucially for a document of this size, margin placement was corrected automatically thanks to Stephen Hicks `marginfix` package.[1] Many figures use the TikZ package for drawing, while inference rules are typeset with Benjamin C. Pierce's `bcprule` package. The SSReflect code listings are prepared with Assia Mahboubi's excellent mode for the `listings` package, included in the SSReflect distribution

[1] This is the Stephen Hicks also responsible for a contribution to the 2008 ICFP contest in TeX.

Preamble

This thesis stems from the experience gained in dealing with a formalization of finite group algebra not only in type theory, but as implemented in the COQ type checker. This manuscript therefore carefully takes into account the fact that both type theory and COQ have evolved simultaneously, but rarely synchronously — meaning that at repeated points in time, the state of development of the prototype did not perfectly match that of the literature. Nonetheless, the goal of reaching a certain level of clarity, and the acknowledgement of considerable progress in the formulations of intuitionistic type theory have made us chose to *not* describe the whole kit and kaboodle found under the hood of COQ. Naturally, time and resources also limited the precision of our understanding of said machinery. Hence, in the following, we have in numerous occasions described only what COQ currently aims at, rather than what it implements. We took great care in mapping out precisely where we strayed, but to best label the engineering corners where we have not managed to dwell as long as the most faithful account would have required, and where we invite but the most stalwart of readers to investigate, we use the icon found in the margin.[2]

2 

This document is not completely self-contained with respect to SSReflect's improvements on the COQ theorem prover. While all theoretical constructions — including small scale boolean reflection — are treated from scratch, the tactic language, or some of the minor syntax extensions made upon COQ's language constructs (irrefutable patterns, `if` notation, etc) are not explained here. We refer the reader to the SSReflect user manual (Gonthier et al. 2008) and to the second section of the tutorial (Gonthier and Mahboubi 2010) Familiarity with at least a first, more "syntax-oriented" version of that tutorial (Gonthier and Le Roux 2009) is assumed.

If you are reading this in dead tree form, you are missing working links within this document (including internal references & back references, links to the bibliography), a few links to resources on the web, and working D.O.I. links in the bibliography.

Introduction

Why I certify

WHY MAKE A COMPUTER CHECK THE PROOF OF A THEOREM OF FINITE GROUP ALGEBRA ?

The introduction is usually the place where the doctoral candidate waxes poetic and dwells too long about the implications of his work. The poetry, we will avoid.

Yet, the Question can not be avoided. The Mathematical Components team has been working towards a COQ formalization of the Feit-Thompson theorem since 2006. It is not finished at the time of this writing. In the meantime, this thesis happened. There must be a reason for convincing a young person to sit down and contribute for a few years of his life.

The first thing one learns of the Feit-Thompson theorem is that its proof is long: the original paper by John Griggs Thompson and Walter Feit is 255 pages long, and is thought to have been the longest of its time (du Sautoy 2008) It also involves mathematics of an unparalleled level in formal reasoning attempts.[3] Is it because its length proves to be a challenge for the social peer review process ? Jean-Pierre Serre voiced that concern nearly a quarter of a century after the proof:

[3] The proof was presented as a two-quarter (24-week) graduate mathematics course at the University of Chicago in 1975 (Bender and Glauberman 1995, p. xi) which hints at the required specialization.

> A more serious problem is the one on the "big theorems" which are both very useful and too long to check (unless you spend on them a sizable part of your lifetime...). A typical example is the Feit-Thompson Theorem : groups of odd order are solvable. (...) What should one do with such theorems, if one has to use them? Accept them on faith? Probably. But it is not a very comfortable situation. (Chong et al. 1986)

However, history shows mathematicians *did* spend on this theorem a sizeable part of their lifetime. The Feit-Thompson proof[4] contained the seed of innovative techniques — what came to be known as "local analysis" — and single-handedly provided enough impetus to greatly increase the number of mathematicians in the area (Scott et al. 2005) Unlike the proof of the Four-colour theorem (Appel and Haken 1977),[5] Feit and Thompson's work did not rely on computer code of debated reliability. Though the mathematical community can sometimes erroneously think results have been proven for years (Lecat 1935), no lingering doubts as to the validity of that paper proof have been voiced.

[4] W. Feit and J. G. Thompson. Solvability of groups of odd order. *Pacific Journal of Mathematics*, 13(3), 1963. URL http://projecteuclid.org/euclid.pjm/1103053941

[5] A proof which led to another COQ formalization effort (Gonthier 2008)

So, why certify a proof that no one doubts is valid ?

Does it have applications? Unlike the verification of a computational tool (Klein et al. 2010; Leroy 2009) or that of COQ's kernel (Barras 1999), completing the check here will not give strong practical or foundational assurances of reliability. Unlike the answers to the POPLMark challenge (Aydemir et al. 2008), this proof will not provide a proof pattern applicable to a recurrent problem of the domain. Also, since they are tailored for finite group theory, proofs of the Mathematical Components team span a much narrower field of mathematics than previous library endeavors (Cruz-Filipe et al. 2004; Rudnicki 2001) — they are the proof of a theorem, first, and a library of formalized algebra, second.

Is it that those proofs will inform mathematical knowledge, and create a long-awaited[6] new understanding of a long and complex proof? Perhaps, but such an impact on finite group theory is not yet obvious, and, should we be skeptical on this sort of outcome, we would be in good company. [7]

[6] The proof was revised in the late 90s, but the revision is but one page shorter than the original (Bender and Glauberman 1995; Peterfalvi 2000).

[7] *"Although this work is purportedly about using computer programming to help doing mathematics, we expect that most of its fallout will be in the reverse direction — using mathematics to help programming computers. [...] In fact, many of our proofs look more like debugger or testing scripts than mathematical arguments. [...] We believe it is quite significant that such a simple-minded strategy succeeded on a "higher mathematics" problem of the scale of the Four Colour Theorem. Clearly, this is the most important conclusion one should draw from this work."* (Gonthier 2005)

If there is one thing that leaves no doubt, it is that, if completed, the COQ proof of the Feit-Thompson theorem will be the largest, most complex piece of formalized mathematics to date — to the point that it has been presented with a mountain peak image on some occasions (Gonthier 2010, for instance). It will, undoubtedly, set a record.

So, why climb such a peak?

To a degree, the question is irrelevant. The reason for the climb moves the climber above anybody else, and often he proves to be an animal of a peculiar breed: asked about his desire for the top in 1923, George Mallory answered with three words that became the most famous in mountaineering (*"Because it's there"*), but without anything remotely like a justification.

A more pertinent approach — for an exterior party — is to notice that nowadays, the leisure climber has a breathable membrane in his jacket, vulcanized rubber on his soles, and chockstones on his rack — and that all were field-developed in exceptional conditions. The pursuit of record-sized proofs presents a challenge that validates scalable, efficient methods, and enforces the improvement of the rest. Before the fame that may come with the eventual completion of the proof, the Feit-Thompson formalization effort is essentially a crucible.

To make this statement more precise, let us go back to what a COQ formalization *is*: it consists in the expression of a mathematical proof in a formal logic, equivalent, thanks to the Curry-Howard isomorphism, to a computer program. Both sides of the isomorphism say exactly the same thing on the purpose of this type of work:

> Tout mathématicien sait d'ailleurs qu'une démonstration n'est pas véritablement "comprise" tant qu'on s'est borné à vérifier pas à pas la correction des déductions qui y figurent, sans essayer de concevoir clairement les idées qui ont conduit à bâtir cette chaîne de déductions de préférence à toute autre.
>
> (Nicolas Bourbaki, L'architecture des mathématiques, In F. Le Lionnais (ed.), *Les grands courants de la pensée mathématique*, Cahiers du sud, 1948)

> The purpose of computing is insight, not numbers.
>
> (Richard W. Hamming, dedication of *Introduction to Applied Numerical Analysis*, 1971)

The most crucial value of the certification of the Feit-Thompson proof therefore has to be found in the actual proof terms, rather than in their type. In that sense, the quotes above remind us that just as when we are writing programs, type-checking is simply an useful auxiliary. It structures the search for proofs — to paraphrase Conor McBride, it makes it easier to search for *good* proofs in the space of *well-typed* proofs, rather than in the space of ASCII lumps.

Why certify? Because the Feit-Thompson type will ask for exceptional programs. And because some could — and perhaps should — become less of an exception.

This thesis

> Generic programming is a programming method that is based in finding the most abstract representations of efficient algorithms. That is, you start with an algorithm and find the most general set of requirements that allows it to perform and to perform efficiently. (A. Stepanov, in Lo Russo 2000)

GENERIC PROGRAMMING OFFERS A WAY TO STRUCTURE PROGRAMMING BY POLYMORPHIC COMPONENTS. It has been purportedly (*ibid.*) developed as a way to emulate the work of mathematicians, who, starting with proofs, end up with axioms characterizing their objects of study. Likewise, the programmer starts with an algorithm, and ends up finding the generic data structure inside. It is then hoped that at the later stage, genericity saves the day by letting the programmer organize algorithms in families of interfaces spanning multiple types.

This thesis deals with how far we can go in this direction. We have mathematical developments to do. It happens that they will be algorithms. How can we leverage COQ's generic programming tools to make writing them easy?

This question presents many challenges. Or, admitting that some requirements of what we are talking about are clear,[8] a single challenge: mathematicians have been blessed with way too good an audience — at least as far as non-elementary mathematics are concerned.

[8] Such as the need for *ad-hoc* polymorphism, which reflects that abstract mathematical statements are expressed on multi-sorted objects ("spanning multiple types") that fit an algebra ("interfaces").

The mathematical audience is expected to carry out part of the abstraction *a posteriori*, understanding while it meets a pattern during proofs, that the reasoning applies in fact to an abstract specification matching several distinct objects.[9] It can also *translate* swiftly between such multi-sorted algebras when the definition of the speaker is equivalent to — yet does not exactly match — the one he knows.

[9] *"La mathématique est l'art de donner le même nom à des choses différentes."* (Henri Poincaré, *Science et méthode*, 1908)

This audience also has a *memory of abstractions*, so that it never forgets if or why a particular mathematical object fits this or that specification. More importantly, when it meets a new object, it can decide upon which abstraction applies by applying the principle of *compositionality* — the idea that the structure of a mathematical object is determined by the meanings of its constituent sub-objects and the rules used to combine them.

Moreover, the mathematical audience has no issue with the extreme fondness mathematicians show for repeatedly sticking a sliver of additional data — and a new name — on a previously-defined abstraction, and merrily car-

rying on. For the astute mathematical audience that is able to *hierarchize specifications*, this is not an all-new environment — just a slight twist on a well-known context.

Having polymorphic interfaces built in the language is not, by itself, enough to emulate all this — so that **our first task** is to understand how to generically program COQ to be a better mathematical audience.

ANOTHER TYPE OF CONCERNS OF OURS is not so much a challenge created by an unfair advantage of mathematicians, but rather an obstacle inherent to our particular context: we are participating to a formalization in *type theory*, in a *programming language*, and among a *multipartite group of two teams working in roughly three locations*. In particular, the concurrent setup, the time scale and the social context in which our formal proof is developed is nothing like the flexibility that Feit and Thompson enjoyed when developing their proof,[10] so that **our second endeavor** is to make sure the facilities for concurrently building large and deep hierarchies of mathematical structures are reliable and efficient.

[10] *"But only Walter [Feit] and I knew just how intertwined our thinking was over a period of more than a year."*
(John Thompson, in Scott et al. 2005)

MOREOVER, A FREQUENT CRITICISM OF FORMAL PROOF is that trusting such a beast hinges not so much on the belief in the type-theoretic technology behind the prover, but rather on the ability to convince oneself that the objects described in the program are truthful, complete representants of the mathematical objects found in a textbook — and more importantly, in the mathematician's head. Said more concisely: program specifications are often unreadable — we mean lemma statements here, not the imperative proofs scripts (that are never read *anyway*). Hence, **our third objective** is to test our generic constructions in a practical setting, paying particular attention to notation facilities of COQ.

ANOTHER SOURCE OF TURMOIL IS TYPE THEORY ITSELF: not only do the mathematics we deal with forgo providing their exact foundations, but they often implicitly consider the concept of set as a centerpiece of their discourse. It is well-known that translating such a discourse in type theory tends to downgrade sets from keystone to brimstone, and we do not expect to make exception with the representation of partial functions — something the native total COQ functions are ill-equipped to represent natively, and therefore **our fourth ordeal**.

ON A BRIGHTER NOTE, THOUGH, IT MAY BE POSSIBLE THAT THE DISCIPLINE OF GENERIC PROGRAMMING LET US APPROACH MATHEMATICAL DEFINITIONS WITH A NEW LOOK, one which would let us select particular ways to exploit polymorphism in the specification of mathematical objects. In particular, we are interested in elementary applications of the notion of relational parametricity, and if there is mathematical sense in favoring functors to represent mathematical objects that are in some sense generic. That **fifth concern** is thus something to be on the lookout for.

FINALLY, OUR MENTIONED INTEREST IN RELATIONAL PARA-

METRICITY MAKES US WONDER IF THEY CAN BE MADE AVAILABLE WITHIN THE CALCULUS OF COQ . The validity of the parametricity theorem for the calculus of COQ notwithstanding, such a theorem can be formalized within COQ for deep embeddings of a smaller calculus. It would be interesting to see, as **a sixth and last path of investigation**, if some usable result for our functorial mathematical objects can be extracted from such a proof.

Overview

THE FIRST CHAPTER is born of the meet between:
- on the one hand, the three tenets of advanced generic programming we require: *specifying* generic interfaces, *abstracting* algorithms and proofs over those specifications, *instantiating* those specifications with particular realizations ;
- and on the other hand, the incarnation of mathematical structures in type theory : *telescopes*.

We explain how in COQ, telescopes are represented using dependent records of a particular sort (§ 1.1.1-§ 1.1.7), and how to assemble them to form compound mathematical objects (§ 1.1.8-§ 1.1.12), with particular interest on a *concept pattern* that lets us implement the three phases above, and, crucially, lets us to do it *a posteriori*, after defining objects (§ 1.1.8).

We then go on explaining how far COQ can go elaborating user input into a typed term (§ 1.2.1-§ 1.2.7), including an ill-known mechanism that occurs at type inference, and tremendously helps the instantiation phase of generic programming (§ 1.2.5): `Canonical Structure`s.

We then reveal that this mechanism is in fact a flavor of the type classes construct of a number of polymorphic programming languages. We explain how this incarnation can help with giving COQ *abstraction memory* (§ 1.3.1), how — with previously-seen record composition paradigms — we can make it *hierarchize definitions* (§ 1.3.2), how it renders a built-in notion of *compositionality* (§ 1.3.3), and remark that it enjoys a nice interplay with COQ's built-in *translation* facilities (§ 1.3.4).

We move on to showing how we made `Canonical Structure`s work measurably better for building *hierarchies*, by changing the way we compose the underlying records, giving rise to the paradigm of Packed Classes (§ 1.4.1). We show that contrary to previous concerns mentioned in the literature, the adoption of this paradigm does not imply a loss of expressivity, treating specifically situations calling for multiple inheritance (§ 1.4.2) and manifest records (§ 1.4.3).

We conclude on the expressivity of `Canonical Structure`s compared to other type class implementations in the wild. Fully leveraging our thorough exposition, we explain how we can make `Canonical Structure`s yield a flavor of *deterministic overlapping instances* (§ 1.4.4) that has been recently suggested as a desirable improvement of that of Haskell, and show how it can be applied to improve the state-of-the art solution to the famous Expression Problem. We finish with a comparison with the `Class` mechanism of COQ (§ 1.4.5), a co-existing, recent implementation of type classes in COQ.

THE SECOND CHAPTER deals firstly with the application of our hierarchy-building paradigm to practical examples, building up to a practical example of how a SSReflect user can pick up a standard release of our library, and certify RSA correct[11] very concisely. We start with exposing the base structures on which the finite group hierarchy is built (§ 2.1.1-§ 2.1.5), porting previous literature on the matter to our Packed Classes paradigm, and providing the guidelines of how, in our context,[12] we understand and write mathematical notions such as intensional sets (§ 2.1.2), functional extensionality (§ 2.1.4), sets (§ 2.1.3), and groups (§ 2.1.5). We then expose personal contributions towards providing the necessary notions (§ 2.2.1) for a smooth, algebraic proof of RSA: the finite number field of prime order (§ 2.2.2), the notion of automorphisms (§ 2.2.4), a number of simple isomorphisms involving cyclic groups and properties of Euler's totient function (§ 2.2.3-§ 2.2.5). We conclude with a small, modular proof of RSA's correctness (§ 2.2.6).

[11] That is, replicate the results of the RSA contribution of COQ.

[12] Despite *looking* like classical reasoning, the SSReflect libraries use no axiom.

This example lets us put in evidence critical fault lines in a well-known paradigm for implementing partial functions with total functions over a type (§ 2.3.1). We explain how on that basis, we managed to improve the general expressivity of our mathematical structure representants by directing type inference using phantom types (§ 2.3.2). We conclude on how that let us redefine partial functions while maintaining strong requirements on the ease of use of those objects (§ 2.3.3-§ 2.3.4).

THE THIRD CHAPTER starts with a personal application of the essence of generic programming to mathematical modelling (§ 3.1): we wrangle a functorial structure out of numerous but generic subgroup definitions that occur throughout the proof of Feit-Thompson (§ 3.1.1), and apply it to simplifying frequent proofs of an elementary property of such subgroups: characteristicity (§ 3.1.2). Then we notice that the generic manipulation of those functors[13] allows us to create links with intensional classes of groups — that is, classes of groups verifying a certain property (§ 3.2). The link, however, does not prove particularly useful until we formalize a richer correspondence with dual classes of groups invariant by isomorphism: torsion theories (§ 3.3). This lets us emulate a proof by isomorphism (§ 3.3.1) on a useful class of properties by studying their value when applied corresponding functorials — despite some issues with force-fitting those highly-polymorphic concepts in COQ (§ 3.3.2). We pursue by noticing the properties of the instances of subgroup-defining functions we defined above hinge on an instance of a "free theorem"[14], and look at how to get it in COQ (§ 3.4.1-§ 3.4.6). After establishing of the link between functoriality and the free theorem in our context (§ 3.4.1), we place what we are looking for within the context of the various existing approaches to parametricity (§ 3.4.2), and develop the key insights on relational parametricity (§ 3.4.3), and suggest a way to get at automatically generated instances of a parametricity theorem based on `Canonical Structure`-based reflection (§ 3.4.4-§ 3.4.6).

[13] Deprecated in group theory since the height of the Soviet era, and only mathematically rediscovered, independently, in the last few years.

[14] P. Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89*, number June, pages 347–359, New York, USA, 1989. ACM Press. ISBN 0897913280. doi:10.1145/99370.99404

Canonical Structures

Tools of the Trade

`Canonical Structure` is the name of a command of the Gallina language[1] that allows a Coq user to equip the unification procedure called by type inference with a specialized heuristic. When asked for a *record* fitting a given *record projection*, this heuristic answers with one of some pre-registered definitions.

[1] Gallina is the name of the specification language of Coq, and the Vernacular is the command language extending Gallina. (Coq 2010, §1–2) lays out its exhaustive documentation.

`Canonical Structure`s also are an instance of a general language construct better known as *type classes* whose claim to fame originated with its implementation in Haskell.

`Canonical Structure`s also are an effective linguistic tool for structuring and organizing proofs — particularly when it concerns hierarchies of objects of some depth.

Finally, knowledge about the organization of such hierarchies can inform `Canonical Structure` use, and influence it .

This chapter aims at explaining, linking, and justifying those four claims. The Coq literature documents the first claim if in a haphazard and fragmented way. The second claim supplies the Coq user with the wide-ranging generic programming facilities of type classes, but has been widely overlooked nonetheless. The third claim was previously made,[2] but only in the framework of the SSReflect libraries. Finally, the fourth claim is entirely original, but its smooth explanation depends on a comprehensive exposition to the three previous ones.

[2] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging Mathematical Structures. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer Berlin / Heidelberg, 2009. doi:10.1007/978-3-642-03359-9_23

1.1 *Model and Implementation: Σ-types and dependent records in* Coq

"The business of abstraction frequently makes things simple."

(Richard W. Hamming, *You and your research.* Talk at Bellcore, 7 March 1986.)

The user of a proof assistant quickly develops sophisticated abstraction needs — usually faster than an alter ego who just wants to compute with some data. Such a user usually means to prove something, and the intuitiveness of abstraction for doing that means that she will probably want to reproduce the generalizations that naturally come to her on the blackboard.

Fortunately, Coq has a module system inherited from the tradition of ML-style languages that purports to do just that: it gives constructs for

specifying the functionality of program components

abstracting programs and proofs over that specification

instantiating programs with specific realizations of these specifications.

In the last couple of decades[3] such a system has proven to be an effective language tool for structuring programs.

[3] D. MacQueen. *Modules for standard ML*. ACM Press, New York, New York, USA, 1984. ISBN 0897911423. doi:10.1145/800055.802036

But it turns out that one does not *have* to use modules. Anyone that has constructor functions that look like `nil` and `cons` can build something that looks like lists.

```
Record seq_type (A: Type): Type
  := mk_seq_type {
    carrier  : Type;
    seq_nil  : carrier;
    seq_cons : A →
      carrier → carrier;
  }.
```

In Coq— since the user is here to prove something — one may want to add an induction principle to that, turning `seq_type` into the definition in Fig. 1.1. One can then define a generic `map` function as follows:

```
Definition map_seq_type (A: Type) (B: Type) (ltA: seq_type A)
  (ltB: seq_type B) (f: A → B): carrier ltA → carrier ltB
:= seq_induction
    ltA
    (fun l      : carrier ltA => carrier ltB)
    (seq_nil ltB)
    (fun a      : A =>
     fun tailA : carrier ltA =>
     fun tailB : carrier ltB =>
     (seq_cons ltB) (f a) tailB).
```

```
Record seq_type (A: Type) : Type
:= mk_seq_type {
  carrier       : Type;
  seq_nil       : carrier;
  seq_cons      : A → carrier →
     carrier;
  seq_induction :
    ∀(P : carrier → Type),
      P seq_nil →
      (∀a: A, ∀l: carrier,
        P l → P (seq_cons a l)) →
      ∀l: carrier, P l
 }.
Implicit Arguments carrier [A].
Implicit Arguments seq_induction [A].
Implicit Arguments seq_nil [A].
Implicit Arguments seq_cons [A].
```

Figure 1.1: A generic list type defined using a record. Implicit arguments enumerated for clarity.

The question of whether to use modules or records to organize a program or development thus comes up frequently. In the world of programming languages, it is a well-known observation that both mechanisms are similar, though by no means interchangeable. Wehr and Chakravarty made this precise in a detailed comparison between the two,[4] and methods of implementing one of the mechanisms using the other abound (Dreyer et al. 2007; Oliveira 2009; Yallop 2007). For proof assistants, the situation is a tad less clear.

[4] S. Wehr and M. Chakravarty. ML Modules and Haskell Type Classes: A Constructive Comparison. In G. Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 188–204. Springer Berlin / Heidelberg, 2008. doi:10.1007/978-3-540-89330-1_14

The `Canonical Structure` mechanism is, historically, an improvement on a development that took records as the main organization tool (Saibi and Huet 2000). It stemmed from the remark that in the phases we distinguished above (on the current page) *abstraction* and *instantiation* were places where automation can help the user.

- The `seq` above was a simple specification with no dependencies but it is nonetheless possible to define structures *depending* on other structures. The type parameter `A` above is a simple variable in the definition above, because we don't need any more primitives to give the type of the list constructors of `seq_type A`. By "dependencies" we mean what happens when we need a record type in place of `A`, to express some of the operations this type comes with. For example, we will see how to specify an order relation as a parametric record in § 1.1.9 on p. 30: if we also want to check this relation is *antisymmetric*, we will have to require the *record parameter* to include a pointer to an *equality* relation in order to write that check.

Thus, a structure comes along with its set of constraints. But at specification time, a user generally wants to manipulate the minimal number of abstractions needed to write his generic proof or program. He rarely wants to mention their prerequisites: in our example, it seems harsh to make him give the whole specification for an equality relation *a second time* when he wants to specify the record type for an order relation. We can therefore benefit from the *inference of structure constraints*.

- It is possible to compose instances of specifications to obtain a new instance of that same structure, or of another. For instance, the (categorial) product of two list types always yields a list type, or one can assemble two singly-linked lists to form a double-linked list. The composition trend can easily run wild enough in a large library that remembering how exactly a complex compound term fits a specification becomes tedious. Here the user benefits from *automatic construction of instances*.

Those two specific improvements upon "the record alternative" represent the essence of a *type class* system. As I will argue based on my experience with the Mathematical Components project,[5] they are a necessity for developing large-scale proof libraries.

[5] http://ssr.msr-inria.inria.fr

On the other hand, we do not mean that "the module alternative" is irrelevant to the world of theorem provers. It is simply less mature. For instance, early developments — either at the dawn of the Mathematical Components libraries, or at the time of Saibi and Huet's development on category theory — had strong but *external* reasons for leaving modules aside.[6]Moreover, recent work makes the implementation of modules in COQ an improving alternative (Soubiran 2010). But I intend to show that advantages such as the automatic generation of instances, seen through the lens of the developer of large libraries of mathematics, make a particularly compelling argument for the use of `Structure`s.

[6] Notably, the absence of said module system, introduced in COQ but in version 7.4. One can also remark that modules in COQ are not first-class, meaning that one cannot instantiate module parameters with variables. And on the other hand, modules have unique facilities for specifying abstraction boundaries using importation and name space, or ascription-based inheritance.

Amokrane Saibi implemented `Canonical Structure`s in COQ version 6.1. A summary of the user-level syntax is present in the manual (Coq 2010, §2.7.15). In a technical description of the unification algorithm used in type inference (Saibi 1999, §4.5,4.7), it is possible to remark facilities for `Canonical Structure` inference, though there is no mention of their operative name or indications on their practical use in that document.

This section aims at describing what this mechanism is, but also at giving some perspective on *where* (on which terms) it operates, and *when* it takes effect. The *where* depends on how COQ represents *dependent records*, and on why they are an encoding of the more general abstraction known as a *telescope*.[7] The *when* requires a reminder on type inference, and how it gives rise to higher-order unification problems. Finally, the section takes special interest in exposing how the user can make a complementary use of the coercion and notation mechanisms of COQ.

[7] N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91(2):189–204, Apr. 1991. ISSN 08905401. doi:10.1016/0890-5401(91)90066-B

1.1.1 Σ-Types and telescopes

The first step to adopt records as an organizational basis is to see how general a construction they are. We want to develop a clear idea of what a COQ record is, and what type-theoretic notion they embody.

In type theory, the construct one should aim for to represent mathematical abstractions is - fortunately - clear. Indeed, said abstractions (usually set-theoretic) are in general composed of a domain, operations on that domain, and, if defined in a proof assistant able to represent them, properties of said operations and domain.

Meanwhile, dependent type theory[8] systematically defines the *dependent product* — a generalization of the usual function space. For instance, in COQ, a term having the type `∀x:A, P(x)` consists in a procedure yielding a proof of `P` for all entries of type `A`. We call those types Π-types or *dependent function types*. Type theories often strive to define a similar generalization for the pair, for terms that consist of a term `x`, *and* the proof that it verifies `P(x)`. Such terms provide a *witness* of a given property, thus correspond to an existential quantification, and carry a Σ-type or *dependent product type*.[9] We will come back later on how COQ represents them exactly.

[8] We assume a basic familiarity with dependent types. Aspinall and Hofmann (2005) provide a good introduction.

[9] Because of the way they map to (respectively) conjunctions or disjunctions through the Curry-Howard correspondence, some call types corresponding to the scheme $\forall x : A.P(x)$ "dependent product types" and $\exists x : A.P(x)$ "dependent sum types". We will try to remove any ambiguity between flavors of labels.

Those dependent pairs are nonetheless sufficient to specify, for example, a type `T` and a binary operation of type `(T → T → T)` on that type: the key of the representation is simply that the type of the second element of the pair depends on the first element. But mathematical structures usually define more than simply one operation. de Bruijn noted that the repetition of that construction was the perfect way to represent mathematical structures, and baptized such an iteration of Σ-types *telescopes* (de Bruijn 1991).

Since then, a number of authors besides de Bruijn have confirmed that telescopes express mathematical structures ideally (Betarte and Tasistro 1998; Mu et al. 2008; Pollack 2000, 2002; Sacerdoti Coen and Tassi 2008), and the claim is now well-established. I add that, **in COQ, one can use records to represent iterated Σ-types, without loss of generality**. It is a frequently encountered but often imprecisely justified claim, especially since neither records nor Σ-types exist as primary constructs in the Calculus of Inductive Constructions, and are instead encoded using inductive types. I therefore support it by providing a formal portrait of COQ's inductives.

1.1.2 The calculus of constructions

In the following paragraphs, I develop just enough of a syntax reminder on the core calculus behind COQ to be able to speak precisely about inductive types, and how they encode both Σ-types and records. To signal where I drop details that fall outside the scope of this document, I proceed incrementally, starting with the now standard syntax of pure type systems (PTS)[10].

[10] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science*, volume 2, chapter 2, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992. ISBN 0198537611

$\mathcal{T}$	::=	$\mathcal{C}$	constant
	|	$\mathcal{V}$	variable
	|	$\mathcal{T}\mathcal{T}$	application
	|	$\lambda\mathcal{V}{:}\mathcal{T}.\mathcal{T}$	abstraction
	|	$\Pi\mathcal{V}{:}\mathcal{T}.\mathcal{T}$	dependent function space

(a)

$\mathcal{E}$	::=	[]	empty environment
	|	$\mathcal{E},\mathcal{V}{:}\mathcal{T}$	variable binding

(b)

Table 1.1: Syntax of a PTS: (a) describes the syntax for terms, (b) the one for environments.

The syntax (Tab. 1.1) of a PTS is that of a λ-calculus parametrized by a specification, *i.e.* sets of sorts, axioms and rules (respectively $\mathcal{S},\mathcal{A},\mathcal{R}$), such

that:[11]

$$\mathcal{S} \subseteq \mathcal{C}$$
$$\mathcal{A} \subseteq \mathcal{C} \times \mathcal{S}$$
$$\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$$

[11] Naturally, the adopted set of constants is technically a parameter of PTS. However, the strength of the theory the user can develop with a PTS varies little with the exact contents of that constant set. Hence, along with the literature on the subject, we consider this fourth parameter implicit in what follows.

$$\frac{c : s \in \mathcal{A}}{\Gamma \vdash c : s}\ (\text{AXIOM}) \qquad \frac{\Gamma \vdash \mathrm{A} : s \qquad s \in \mathcal{S}}{\Gamma, x : \mathrm{A} \vdash x : \mathrm{A}}\ (\text{START})$$

$$\frac{\Gamma \vdash \mathrm{A} : \mathrm{B} \qquad \Gamma \vdash \mathrm{C} : s}{\Gamma, x : \mathrm{C} \vdash \mathrm{A} : \mathrm{B}}\ (\text{WEAKENING}) \qquad \frac{\Gamma \vdash \mathrm{F} : (\Pi x : \mathrm{A} . \mathrm{B}) \qquad \Gamma \vdash a : \mathrm{A}}{\Gamma \vdash \mathrm{F}a : \mathrm{B}[x \mapsto a]}\ (\text{APPLICATION})$$

$$\frac{\Gamma, x : \mathrm{A} \vdash b : \mathrm{B} \qquad \Gamma \vdash (\Pi x : \mathrm{A} . \mathrm{B}) : s}{\Gamma \vdash (\lambda x : \mathrm{A} . \mathrm{B}) : (\Pi x : \mathrm{A} . \mathrm{B})}\ (\text{ABSTRACTION})$$

$$\frac{(s_1, s_2, s_3) \in \mathcal{R} \qquad \Gamma \vdash \mathrm{A} : s_1 \qquad \Gamma, x : \mathrm{A} \vdash \mathrm{B} : s_2}{\Gamma \vdash (\Pi x : \mathrm{A} . \mathrm{B}) : s_3}\ (\text{PRODUCT})$$

$$\frac{\Gamma \vdash \mathrm{A} : \mathrm{B} \qquad \Gamma \vdash \mathrm{B}' : s \qquad \mathrm{B} =_\beta \mathrm{B}'}{\Gamma \vdash \mathrm{A} : \mathrm{B}'}\ (\text{CONVERSION})$$

Figure 1.2: Typing of the PTS $(\mathcal{S}, \mathcal{A}, \mathcal{R})$

We let $x, a, \mathrm{A}, \mathrm{B}, \mathrm{C}$ range over $\mathcal{V}$, c range over $\mathcal{C}$, $s, s_1, \ldots, s_n$ range over $\mathcal{S}$, and Γ range over $\mathcal{E}$. Fig. 1.2 lays out the typing judgment associated to that calculus.[12] We assume standard definitions of the free variables of a term, and that $t[x \mapsto u]$ where t, u are terms and x a variable, denotes the usual notion of capture-avoiding syntactic substitution of x by u in t.

[12] As customary, in the non-dependent cases, we will contract the type of the product in the abstraction rule into the more compact $\mathrm{A} \to \mathrm{B}$.

This generic syntax encompasses the description of well-known calculi: all the systems of the Barendregt cube are retrievable from specifications using $\mathcal{S} = \{\texttt{Set}, \texttt{Type}\}$, $\mathcal{A} = \{\texttt{Set} : \texttt{Type}\}$ and rules of the form:

$$\mathcal{R}_\lambda = \{(\texttt{Set}, \texttt{Set}, \texttt{Set})\} \quad \text{which give the } \lambda\text{-calculus}$$
$$\mathcal{R}_\mathrm{F} = \mathcal{R}_\lambda \cup \{(\texttt{Type}, \texttt{Set}, \texttt{Set})\} \quad \text{which give system F}$$
$$\mathcal{R}_{\mathrm{F}_\omega} = \mathcal{R}_\mathrm{F} \cup \{(\texttt{Type}, \texttt{Type}, \texttt{Type})\} \quad \text{which give system } \mathrm{F}_\omega$$
$$\mathcal{R}_{\mathrm{CC}} = \mathcal{R}_{\mathrm{F}_\omega} \cup \{(\texttt{Set}, \texttt{Type}, \texttt{Type})\} \quad \text{which give CC}$$
$$\mathcal{R}_\mathrm{P} = \mathcal{R}_\lambda \cup \{(\texttt{Set}, \texttt{Type}, \texttt{Type})\} \quad \text{which give } \lambda\text{LF}$$

1.1.3 The calculus of constructions with universes

Let us now view the transitive closure of the typing relation defined by $\mathcal{A}$ as an inclusion: say that T *contains* U if $\mathrm{U} : \mathrm{T} \in \mathcal{A}$ or if there is a V containing U such that $\mathrm{V} : \mathrm{T} \in \mathcal{A}$. This inclusion relation defines a *universe hierarchy* with "smaller" sorts at the *bottom*.

For logicians, a classic line of enquiry consists in comparing set and type theory proofs, and it has led to specific conditions under which a sort can *contain* another. Of particular interest is the notion of predicativity:[13] if an application of the PRODUCT rule above quantifies universally over a sort containing the type it is defining, we say that this quantification is *impredicative*. This occurs for example with System F, and calculi of which it is a subsystem.

[13] A treatment of this issue falls beyond the scope of the document. We refer the reader to Bertot and Castéran (2004, §4.3) for a fair introductory discussion in the framework of COQ.

The bottom of the universe hierarchy in COQ is in fact more complex, for historical reasons: before addressing the question of predicativity, concerns related to extraction led Christine Paulin to split the lowest sort of the hierarchy into `Prop` and `Set` - at the time both impredicative. The first represented proofs and was used in an opaque manner, with COQ prohibited from unfolding the computational content of such a term. The extraction mechanism could forget their definition in a computationally-safe manner. The second corresponded to more ordinary programs, preserved by extraction. Both were included in the higher sort `Type`.

Much of this layout remains: `Set` and `Prop` are still the smallest sorts, both in $\texttt{Type}_0$. However, the behavior of `Set` is now predicative, and its continued existence is thus only justified by non-meta-theoretic concerns — namely, the extraction mechanism and backwards-compatibility with an `impredicative-set` switch allowing the type-checker to go back to impredicative reasoning. With theoretical concerns in mind, it is perfectly safe to replace all instances of `Set` by `Type`, so that I ignore the former for the rest of this.

The Calculus of Constructions with Universes (CC_ω) was first introduced by Coquand (1986) to provide an extension of CC with tightly controlled impredicativity. It has an infinite hierarchy of sorts $\texttt{Type}_0, \texttt{Type}_1, \ldots$ such

that:

$$\mathcal{S} = \{\texttt{Prop}\} \cup \{\texttt{Type}_i \mid i \in \mathbb{N}\}$$
$$\mathcal{A} = \{\texttt{Prop}:\texttt{Type}_0\} \cup \{\texttt{Type}_i:\texttt{Type}_{i+1} \mid i \in \mathbb{N}\}$$
$$\mathcal{R} = \begin{Bmatrix} (\texttt{Prop}, \texttt{Type}_i, \texttt{Type}_i) & i \in \mathbb{N} \\ (s, \texttt{Prop}, \texttt{Prop}) & s \in \mathcal{S} \\ (\texttt{Type}_i, \texttt{Type}_j, \texttt{Type}_{\max(i,j)}) & i,j \in \mathbb{N} \end{Bmatrix}$$

And indeed, if one but syntactically renames our previous impredicative sort, `Set`, into `Prop`, he remarks that CC is a sub-system of CC_ω. However, the latter strictly limits impredicativity to a special sort: one can form propositions by quantifying over all propositions, using rule $(\texttt{Type}_i, \texttt{Prop}, \texttt{Prop})$, but we now use the smallest sort on which quantification is predicative, $\texttt{Type}_0$, for all terms, ordinary or polymorphic.

1.1.4 The cumulative calculus of constructions with universes

A consequence of the product rule seen in CC_ω is that a well-typed proposition is always expressed at a given universe level rather than over all known universes. Moreover, the COQ compiler forgoes the fixed universe labelling of CC_ω, and instead implements a constraint-generation algorithm that allows it to fix the lowest universe of the terms it encounters as late as possible.[14]

To move closer to the (relative) leniency of this implicit universe checking, we extend the calculus with a cumulativity relation that better explains the constraints of the algorithm, which we plug in the CONVERSION rule.

We henceforth denote the β-reduction by $\triangleright_\beta$[15], and the *conversion* relation $\simeq_\beta$ as its reflexive, symmetric, transitive closure. The *cumulativity* relation $\preceq$ is the transitive closure of the partial relation defined by:

- $t_1 \preceq t_2$ if $t_1 \simeq_\beta t_2$
- $\texttt{Prop} \preceq \texttt{Type}_n$ for all $n \in \mathbb{N}$
- $\texttt{Type}_n \preceq \texttt{Type}_m$ for all $(n,m) \in \mathbb{N} \times \mathbb{N}$ with $n \leq m$
- $\Pi x : t_1 \,.\, t_1' \preceq \Pi x : t_2 \,.\, t_2'$ for any t_1, t_2, t_1', t_2' such that $t_1 \simeq_\beta t_2$ and $t_1' \preceq t_2'$

The rule CONVERSION of Fig. 1.2 on the preceding page then becomes:

$$\frac{\Gamma \vdash t_1 : t_3 \qquad \Gamma \vdash t_2 : \sigma \qquad t_3 \preceq t_2}{\Gamma \vdash t_1 : t_2}\ (\text{CUMULATIVE-CONVERSION})$$

Figure 1.3: Conversion rule of a PTS extended with cumulativity

Armed with this extension, we can reformulate the calculus once again. The set of rules governing the product becomes:

$$\mathcal{R} = \begin{Bmatrix} (\texttt{Prop}, \texttt{Type}_i, \texttt{Type}_i) & i \in \mathbb{N} \\ (s, \texttt{Prop}, \texttt{Prop}) & s \in \mathcal{S} \\ (\texttt{Type}_i, \texttt{Type}_j, \texttt{Type}_k) & i,j \in \mathbb{N}, \{i,j\} \leq k \end{Bmatrix}$$

The introduction of a cumulativity relation is a classic extension of predicative calculi with an infinite hierarchy of universes. It is perhaps most prominently featured in Luo's *ECC* (Luo 1994), but that calculus contains other extensions — such as Σ-types — that we will approach differently here. We instead refer the reader to Courant (2002) – what we present at this stage is ECC^-, disguised as "CC_w^+" –, which is without a doubt its most incremental treatment, closely followed by the work of Barras on *Cumulative Type Systems* (CTS) (Barras 1999; Barras and Grégoire 2005).

[14]

[15] The definition will come in Tab. 1.4 on p. 25. It holds no surprises

1.1.5 *The cumulative calculus of inductive constructions with universes*

The addition of inductive types provide a primitive construct for what was previously defined using polymorphic type constructors and impredicative encodings. Indeed, reasoning with those "encoded" inductives does not allow to derive the validity of their induction principle as stated using dependent types (Geuvers 2001) — which indicates that those encodings leave the theory bereft of a crucial tool, and prompts the addition of native inductives types. This comprises of the addition of constructors defining those inductive types (their introduction rules), case elimination providing access to their components, and a recursion operator.

The goal of this subsection is to make a clear depiction of non-recursive inductives with a single constructor, so that I make the deliberate choice of referring to the literature on topics that step even slightly over this editorial bee-line.

We augment the syntax of the calculus with the constructs of table 1.2. Some examples of terms generated by this syntax figure in table 1.3. We start by noticing that we access names contained in an inductive declaration by using object-style dot notation. We usually partition the definition of inductive arguments into polymorphic parameters of the definition, which are global to the definition and to which the constructor can refer, and "real" arguments, whether of constructors or inductives. The parameters occur both in the inductive type and constructor applications, while their concrete arguments are distinct.[16]

[16] The expressivity boon obtained by distinguishing generic parameters from indices of an inductive family is well defended in (Dybjer 1991, §7).

As is often done in physics, we favor emboldened characters for representing vectors, and use arrows only when strictly necessary: $\overrightarrow{\mathbf{x}}$ thus stands for a two-dimensional (non-necessarily square) matrix whose i^{th} row vector is $\mathbf{x_i}$. The jth component of the latter is $x_{i,j}$, and its length is $|\mathbf{x_i}|$. We will assume operator symbols used in that notation to mean the distributed version of the unary equivalent, e.g. $\mathbf{x}:\mathbf{A}$ signifies $x_1:\mathrm{A}_1, \ldots, x_n:\mathrm{A}_n$

Then for the case construct, we present an example of a term e of some inductive type $(\mathrm{I}\,\mathbf{p}\,\mathbf{u})$, with $\mathbf{p}$ the polymorphic parameters of the constructor for that type and $\mathbf{u}$ some additional arguments applied to the formed inductive $\mathrm{I}\,\mathbf{p}$. The reduction to branch h_j of the pattern-match is naturally - as in all lines of this table - expected to yield the stated term when term application goes over well arity-wise, namely when $|\mathbf{p}| = |\mathbf{q}|$. We assume that the n-ary extension of term substitution is parallel.

Finally, we look at a fixpoint $\mathrm{fix}_n\{f:\mathrm{T} \coloneqq \mathrm{M}\}$, whose reduction is subject to a guard condition that ensures strong normalization of the calculus by verifying that unfolding the fixpoint terminates. That condition is strongly dependent on n, the index of the argument of f on which recursion is structural. We use the GUARDED keyword without further explanation, and invite the reader to find details in Giménez (1995).

$\mathcal{T}$	::=	$\ldots$	(see 1.1(a))
	|	$\mathrm{Ind}\{\mathcal{V}:\mathcal{T} \coloneqq \overrightarrow{\mathcal{V}:\mathcal{T}}\}.\mathcal{V}$	inductive/constructor name
	|	$\langle \mathrm{P} \rangle$ case $\mathcal{T}$ of $\overrightarrow{\mathcal{V} \Rightarrow \mathcal{T}}$	pattern-matching
	|	$\mathrm{fix}_n\{\mathcal{V}:\mathcal{T} \coloneqq \mathcal{T}\}$	recursion

(a)

$\mathcal{E}$	::=	$\ldots$	(see 1.1(b))
	|	$\mathcal{E}, \mathrm{Ind}\{\mathcal{V}:\mathcal{T} \coloneqq \overrightarrow{\mathcal{V}:\mathcal{T}}\}$	inductive declaration

(b)

Table 1.2: Additional terms of the Calculus of Inductive Constructions. We also extend contexts with the traditional signature (b) of inductive declarations, that parametrizes the typing judgment.

We finally add the typing rules of Fig. 1.4 on p. 25 to those of 1.2 on p. 21. Note that in the CONSTR and CASE rule, A is a product type admitting at least $\mathbf{u}$ arguments. Moreover, constructors store a copy of the parametric arguments of an inductive type, and in the elimination form, the parametric arguments appear again.

Grammar rule example	Coq term example	Reduces to	Note

$\mathrm{Ind}\{\mathrm{I} : \Pi\, \mathbf{q}{:}\mathbf{Q}\,.\, s \coloneqq \overrightarrow{\mathrm{C}{:}(\Pi\, \mathbf{q}{:}\mathbf{Q}\,.\,\Pi\, \mathbf{v}{:}\mathbf{V}\,.\,\mathrm{I}\mathbf{q})}\}$	`Inductive I(q:Q) : s := {`
	 `... }` $i-1$ constructors
	 C_i`:forall(q:Q),`
	 `forall(v:V),Iq;`
	 `...`
	 `}.`
			(in the future, we will often take $\mathrm{I}\mathbf{q}$ to be a product type itself and C_i to return some fuller application $\mathrm{I}\mathbf{q}\mathbf{u}$)

$\mathrm{Ind}\{\mathrm{I}{:}\Pi\, \mathbf{q}{:}\mathbf{Q}\,.\, s \coloneqq \mathbf{C}{:}\mathbf{T}\}.\mathrm{I}$	`I`		the family name
$\mathrm{Ind}\{\mathrm{I}{:}\Pi\, \mathbf{q}{:}\mathbf{Q}\,.\, s \coloneqq \mathbf{C}{:}\mathbf{T}\}.\mathrm{C}_i$	$\mathtt{C}_i$		a constructor name

$\langle \lambda \mathbf{u}{:}\mathbf{U}\,.\,\lambda y{:}\mathrm{I}\mathbf{p}\mathbf{u}\,.\,\mathrm{P}_0\rangle$ case e of $\overrightarrow{\mathbf{v} \Rightarrow h}$	`match` e `as` x `in (I_u)`
	 `return` P_0
	 `with`
	 `| ... }` $j-1$ branches
	 `|` $(\mathrm{C}_j\mathbf{p}\mathbf{a}$`=>`$h_j\mathbf{a})$
	 `| ...`
	 `end`
		$h_j[\mathbf{v} \mapsto \mathbf{a}]$	when $\begin{cases} e{:}(\mathrm{I}\mathbf{p}\mathbf{u}) \\ \mathrm{C}_j{:}\Pi\,\mathbf{q}{:}\mathbf{Q}\,.\,\Pi\,\mathbf{v}{:}\mathbf{V}\,.\,\mathrm{I}\mathbf{q}\mathbf{u} \end{cases}$

$\mathrm{fix}_n\{f{:}\Pi\mathbf{u}{:}\mathbf{U}\,.\,\mathrm{V} \coloneqq \mathrm{M}\} = \mathrm{F}$	`fix f (u:U)`
	 `{struct u`$_n$`}: V :=`
	 `M.`
		$\mathrm{M}[f \mapsto \mathrm{F}]$	when GUARDED(F,n)

Table 1.3: Rule examples, Coq equivalents and intended reductions for terms of the grammar in Tab. 1.2 on p. 23.

Beyond the previously mentioned GUARDED condition, we use two clauses ELIM and INDUC. The latter checks that an inductive declaration is well-formed, to ensure strong normalization and consistency, including the famous *strict positivity* condition. We do not detail it here, and refer the reader to Paulin-Mohring (1996, §III.3.3) for details.

The side condition ELIM occurs in rule CASE, which is the most complicated. The rule CASE aims at typing a pattern matching case with the predicate that labels it. This predicate is, incidentally, only necessary to deal with cases of dependent elimination where type checking could turn out to be undecidable - as readers familiar with the `return` keyword in Coq will have noticed. Again, we refer to Paulin-Mohring (1996, §III.3.5) for details. The intent is that once the type of this predicate instantiates with the particular instance of a constructor found in the matched term, it will yield a sort, that will be compared with the inductive to restrict the class of object that can be built by pattern matching. We have already touched on that subject — predicativity — on p. 21.

In the CIC, it is customary to distinguish the admissible reduction operations and to give them distinct greek letters. We therefore expose conversion in table 1.4 on the facing page, where declarations are parametrized by the context and the global environment $\mathcal{E}$, witnessing global declarations. Readers wishing for a more complete refresher will consult (Coq 2010, §4.3).

Naturally, the calculus we present here is still far from representing either all features of the CIC, or all the nuances of the select few we mention. Among the most outrageously blatant omissions is a more complete treat-

Rule name	Definition	Intended meaning
β-reduction	$\mathcal{E}[\Gamma] \vdash (\lambda x{:}T\,.\,t)u \rhd_\beta t[x \mapsto u]$	vanilla β-contraction
ι-reduction	See pattern-matching in table 1.3	Inductive elimination
δ-reduction	$\mathcal{E}[\Gamma] \vdash x \rhd_\delta t$ if $(x := t : T) \in \Gamma$	meta-expansion of definitions in context or
	$\mathcal{E}[\Gamma] \vdash c \rhd_\delta t$ if $(c := t : T) \in \mathcal{E}$	environment
ζ-reduction	$\texttt{let}\ x\ \texttt{:=}\ u\ \texttt{in}\ t \rhd_\zeta t[x \mapsto u]$	declaration-destroying meta-expansion of local definitions

Table 1.4: Conversion rules of the Calculus of Inductive Constructions.

$$\frac{\mathrm{WF}(\Gamma) \qquad \textsc{induc}\{(\mathrm{I}{:}\mathrm{A}), \mathbf{C}{:}\mathbf{T}\} \qquad \Gamma \vdash \mathrm{A}{:}s \qquad \forall(\mathrm{C}_i : \mathrm{T}_i),\quad \Gamma,(\mathrm{I}{:}\mathrm{A}) \vdash \mathrm{T}_i{:}s_i}{\mathrm{WF}(\Gamma, \mathrm{Ind}\{\mathrm{I}{:}\mathrm{A} \coloneqq \mathbf{C}{:}\mathbf{T}\})}\ (\text{IND-WF})$$

$$\frac{\mathrm{WF}(\Gamma) \qquad \mathrm{Ind}\{\mathrm{I}{:}\mathrm{A} \coloneqq \mathbf{C}{:}\mathbf{T}\} \in \Gamma}{\Gamma \vdash \mathrm{Ind}\{\mathrm{I}{:}\mathrm{A} \coloneqq \mathbf{C}{:}\mathbf{T}\}.\mathrm{I}{:}\mathrm{A}}\ (\text{IND})$$

$$\frac{|\mathbf{p}| = |\mathbf{q}| \mathrm{WF}(\Gamma) \qquad \mathrm{Ind}\{\mathrm{I}{:}\Pi\,\mathbf{q}{:}\mathbf{Q}\,.\,\mathrm{A} \coloneqq \mathbf{C}{:}\mathbf{T}\} \in \Gamma}{\Gamma \vdash \mathrm{Ind}\{\mathrm{I}{:}\Pi\,\mathbf{q}{:}\mathbf{Q}\,.\,\mathrm{A} \coloneqq \mathbf{C}{:}\mathbf{T}\}.\mathrm{C}_i\,(\mathbf{p}\,\mathbf{a}){:}\mathrm{I}\,\mathbf{p}\,\mathbf{u}}\ (\text{CONSTR})$$

$$\frac{\mathrm{Ind}\{\mathrm{I}{:}\Pi\,\mathbf{q}{:}\mathbf{Q}\,.\,\mathrm{A} \coloneqq \mathbf{C}{:}\mathbf{T}\} \in \Gamma \qquad |\mathbf{p}| = |\mathbf{q}| \qquad \Gamma \vdash \mathrm{P}{:}\mathrm{B} \qquad \textsc{elim}(\mathrm{I}\,\mathbf{p}{:}\mathrm{A}\,;\,\mathrm{B}) \qquad \Gamma \vdash e{:}\mathrm{I}\,\mathbf{p}\,\mathbf{u} \qquad \forall(\mathrm{C}_i : \overbrace{\Pi\,\mathbf{q}{:}\mathbf{Q}\,.\,\Pi\,\mathbf{v}{:}\mathbf{V}\,.\,\mathrm{I}\,\mathbf{q}\,\mathbf{w}}^{=\mathrm{T}_i}),\ \Gamma \vdash h_k : \Pi\,\mathbf{v}{:}\mathbf{V}\,.\,\mathrm{P}\,\mathbf{w}(\mathrm{C}_i\,\mathbf{p}\,\mathbf{v})}{\Gamma \vdash \langle \mathrm{P} \rangle\ \mathrm{case}\ e\ \mathrm{of}\ \mathbf{h}{:}\mathrm{P}\,\mathbf{u}\,e}\ (\text{CASE})$$

$$\frac{\Gamma;(f{:}\mathrm{T}) \vdash \mathrm{M}{:}\mathrm{T} \qquad \textsc{guarded}(\mathrm{fix}_n\{f{:}\mathrm{T} \coloneqq \mathrm{M}\})}{\Gamma \vdash \mathrm{fix}_n\{f{:}\mathrm{T} \coloneqq \mathrm{M}\}{:}\mathrm{T}}\ (\text{FIX})$$

Figure 1.4: Additional typing rules for inductives of the Calculus of Inductive Constructions

ment of conversion, followed by elements such as let polymorphism, modules or coinductive types. In this alternative presentation, we have striven to give a minimalistic description, and to stay as close as possible to the calculus of Lee and Werner (2011) — specifically, we omitted a judgmental equality which matters little here, let assignments, and syntax details for *mutual* induction or recursion — which is lately considered to be closest to the modern implementation of Coq (Herbelin 2009, compare Coq 2010, §4). This implied steering away from Coq models based on strong elimination (Miquel 2001; Werner 1994), and bypassing contravariant subtyping. We refer the reader eager for a complete description to the (other) authoritative landmarks of the Coq literature (Barras 1999; Cornes 1997; Letouzey 2004; Paulin-Mohring 1996). Those references also contain extensive studies of the meta-theory of such a calculus.

1.1.6 Inductive representations of telescopes

As we previously mentioned, type theories (Luo 1994; Martin-Löf 1984) define Σ-types as primitive constructs. Luo, for example, introduces them with the following rules:

$$\frac{\Gamma \vdash \mathrm{A}{:}\texttt{Type}_i \qquad \Gamma, x{:}\mathrm{A} \vdash \mathrm{B}{:}\texttt{Type}_i}{\Gamma \vdash \Sigma\, x{:}\mathrm{A}\,.\,\mathrm{B}{:}\texttt{Type}_i}\ (\Sigma\mathrm{T})$$

$$\frac{\Gamma \vdash \mathrm{M}{:}\mathrm{A} \qquad \Gamma \vdash \mathrm{N}{:}\mathrm{B}[x \mapsto \mathrm{M}] \qquad \Gamma, x{:}\mathrm{A} \vdash \mathrm{B}{:}\texttt{Type}_j}{\Gamma \vdash \langle \mathrm{M}, \mathrm{N} \rangle_{\Sigma x:\mathrm{A}\,.\,\mathrm{B}} : \Sigma\, x{:}\mathrm{A}\,.\,\mathrm{B}}\ (\textsc{pair})$$

Figure 1.5: Introduction rules for Σ-types in Luo's ECC. The annotation of the pair is necessary to disambiguate type inference, but we do not dwell on it since this problem is of little concern here.

Additionally, they come with projections π_1,π_2 that allow the user to access right and left member of those dependent pairs. Naturally, those rules relate to corresponding language constructs in the implementations of those type theories, such as in LEGO.

Those rules pose no problem by themselves in ECC, since they come with the limitation that Σ-types be *non-propositional types*. Trouble occurs when one wants to extend a rule such as ΣT to produce proposition types, because of the *impredicative* behavior of `Prop` in ECC. Indeed, since large Σ-types and impredicativity are inconsistent,[17] it is impossible to introduce propositional Σ-types for which the sort assigned to the quantified variable (A in rule ΣT) is unconstrained.The best logically consistent addition one can make to the calculus is called *small* Σ-types, a rule which adds the constraint that the quantified type variable be of sort `Prop`:

[17] J. G. Hook and D. J. Howe. Impredicative Strong Existential Equivalent to Type:Type. Technical report, Cornell University, Ithaca, NY, USA, 1986. URL http://www.ecommons.cornell.edu/handle/1813/6600

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

To reproduce native Σ-types at the type and proposition level in COQ, we would have to introduce new rules, along with sort restrictions and projectors that verify that the quantification in a Σ-type is predicative (akin to the rule set $\mathcal{R}$ binding the product).

Fortunately, we already have introduced a general-purpose boxing construct, along with projectors verifying strict destruction preconditions:[18] inductive types. Using those *recursive types with constructors* as a representation for *existential types* just amounts to making a well-known correspondence between the two explicit.[19] In COQ a Σ-type is therefore introduced as an instance of the following type:

[18] Represented by ELIM in Fig. 1.4 on the previous page.

[19] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(3):470–502, 1988. doi:10.1145/44501.45065

```
Inductive ex (A:Type) (P:A → Prop) : Prop :=
ex_intro : ∀x:A, P x → ex (A:= A) P.
```

It simply consists in a non-recursive inductive with a universal quantification on a variable (the *witness*) and a proof, both of which do not appear in the final type. Notice that at the definition stage, the inductive is just a boxing construct: is in particular possible to construct an object of type `(ex P)`, with `P` of type `(A → Prop)` for an `A` of any sort, including $\texttt{Type}_i \succeq \texttt{Prop}$. However, accessing the computational content of such an inductive is impossible, as shown in Fig. 1.6.

```
Variable A : Type.
Variable P : A → Prop.
Definition f (A: ex P) : Prop.
 case.
> Case analysis on sort Type is
> not allowed for inductive
> definition ex.
```

(a)

```
Definition f (z: ex P) : A :=
  match z with
   |ex_intro x P = > x
  end.

> Incorrect elimination of "z" in the
> inductive type "ex": the return type
> has sort "Type" while it should be
> "Prop".
> Elimination of an inductive object
     of
> sort Prop is not allowed on a
     predicate
> in sort Type because proofs can be
> eliminated only to build proofs.
```

(b)

Figure 1.6: Invalid attempt to destruct an existential type. Notice the difference in clarity of the error messages.

Naturally, in a context where we can stay impredicative, such as when building a proof (instead of a `Definition`), case analysis succeeds without incident. This suffices to avoid incoherences, since accessing the witness of an existential is vital for the encoding of paradoxes entailed by large Σ-types (Paulin-Mohring 1996, §III.4.1.2).

If piggy-backing on inductive types is the right way to represent Σ-type, we said nothing of *iterating* that Σ-type construction to form telescopes yet. The COQ standard library offers us a suggestion, however:

```
Inductive ex2 (A:Type) (P Q:A → Prop) : Prop :=
ex_intro2 : ∀x:A, P x → Q x → ex2 (A:= A) P Q.
```

What we require is a sequence of dependent products that allows, informally speaking, the representation of rich type contexts as ‘objects’: the

first term limits the scope of the following ones, justifying the name *telescope*. But there is no need to add unnecessary boxing levels to represent iterated Σ-types. By curryfication, we can simply use the unlimited arity of the dependent function type which constitutes the *type* of an inductive *constructor* to provide a dependent product of unlimited size, thus storing all the elements of a telescope.

This suggests that, in COQ, the best possible representation of telescopes we can craft is as non-recursive inductive types with a single constructor.

Let us consider a general telescope, *i.e.* a nested Σ-type (as reminded in § 1.1.1 on p. 19):

$$\Sigma x_1{:}\mathrm{A}_1 \,.\, (\Sigma x_2{:}\mathrm{A}_2 \,.\, (\ldots(\Sigma x_n{:}\mathrm{A}_n \,.\, \mathrm{B})\ldots)$$

To avoid "greek buffer overflow", we will note this type with brackets, as such:

$$[x_1{:}\mathrm{A}_1, \ldots, x_n{:}\mathrm{A}_n]\mathrm{B}$$

Using our emboldened vectorial notation this is more succinctly represented as $[\mathbf{x}{:}\mathbf{A}]\mathrm{B}$.

It thus corresponds in COQ to the inductive:

```
Inductive myTele : Type :=
  Build_myTele : (x_1:A_1) → (x_2:A_2) → ... → B → myTele.
```

Indeed, the limitation introduced by having only small Σ-types weighs little in our setting: our intent is to define structures for programming, computing, and proving theorems coming from the general mathematical literature. Most of the Σ-types we intend to use are therefore non-propositional, and moreover, since they come from pages of mathematics grounded in ZFC,[20] they follow set-theoretic rules regarding the objects they can contain.

Nevertheless, those dependent constructs jump `Type` levels at each boxing stage — which implies some limitations, particularly when using them in mathematical contexts which involve large types. Since the inductive resolves at declaration time as attaining a higher kind than that of its arguments, it raises universe inconsistencies when applied to itself — even when this application involves no recursion[21] — owing to the not-quite-modular universe checking algorithm of COQ (Herbelin 2005): Saibi comes up on it in his study of category theory when he tries to define functor categories, for example (Saibi 1999, §8.7). The same sort of problem has also come up in recent developments using inductives in the same fashion (Krishnaswami et al. 2008; Spitters and van der Weegen 2011; Verbruggen et al. 2008), and prevents the adaptation of some nice techniques to COQ (Capretta 2004).

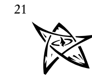
The procedure for universe checking in COQ is currently under improvement, in part to address those concerns.

The formal meaning of this bracket notation is inductively that $[x:\mathrm{A}]\mathrm{B}$ is $\Sigma x{:}\mathrm{A}.\mathrm{B}$, and, for all n, $[x_1{:}\mathrm{A}_1, \ldots, x_{n+1}{:}\mathrm{A}_{n+1}]\mathrm{B}$ is $[x_1{:}\mathrm{A}_1, \ldots, x_n{:}\mathrm{A}_n](\Sigma x_{n+1}{:}\mathrm{A}_{n+1} \,.\, \mathrm{B})$.

It acknowledges that we have arbitrarily presented Σ-type iteration as right-associating nesting. If this associativity choice does have some consequence for defining primitive records (Pollack 2000, 2002), inductive representations do not carry such a slant.

Unfortunately this notation blurs something that the successive right-associative dots of its meta-expansion hinted at: any expression A_j may depend on all x_i, $i \leq j$.

[20] A. A. Fraenkel, Y. Bar-Hillel, and A. Lévy. *Foundations of set theory*. Studies in logic and the foundations of mathematics. Noord-Hollandsche U.M., 1973. ISBN 9780720422702

[21] 

1.1.7 *Dependent records as inductive types*

We can now summarize the translation from a given abstract telescope down to a COQ record. Suppose we want to represent a context in which we

have an order relation, as given by a carrier type, and an antisymmetric, transitive, binary comparison operator.

The corresponding telescope has the shape:

```
[carr:Type, leq:carr → carr → Prop, P1:asym(leq)] P2:trans(leq)
```

In Coq, we have already established this can best be represented as a non-recursive singleton inductive type:

```
Inductive orderedType : Type :=
  OrderedType :
    ∀(carr: Type)
    (leq : carr → carr → Prop)
    (P1: asym carr leq)
    (P2: trans carr leq),
    orderedType.
```

As expected:

```
Definition asym (T:Type)
     (o: T → T → Prop) : Prop :=
  ∀(x y : T),
    (o x y) → (o y x) → x = y.

Definition trans (T:Type)
      (o: T → T → Prop) : Prop :=
  ∀(x y z : T),
    (o x y) → (o y z) → (o x z).
```

Figure 1.7: Antisymmetry and transitivity definitions for a propositional order relation.

Since this restricted kind of inductive is particularly common it has its own *syntactic sugar*, which embodies the equation "telescope = record":

```
Record orderedType : Type := OrderedType {
  carr : Type;
  leq  : carr → carr → Prop;
  P1   : asym carr leq;
  P2   : trans carr leq
}.
```

We use CamelCase for identifiers, and conventionally capitalize the first letter of constructors only. On the other hand, we favor ML-style underscores for lemmas and proofs.

This macro also defines the *projections* `carr`, `leq`, `P1` and `P2`, that give access to *members* of such a *record*, in the same scope as the record definition, just as it does with an inductive (Fig. 1.8): indeed, those projections are a simple application of pattern-matching.

```
carr : orderedType → Type
leq  : ∀(x:orderedType),
         carr x → carr x → Prop
P1   : ∀(x:orderedType),
         asymm (carr x) (leq x)
P2   : ∀(x:orderedType),
         trans (carr x) (leq x)
```

Figure 1.8: Example of the types of record projections generated by Coq

The (trivial) recursion and induction principle are also generated as usual. The details are in Coq (2010, §2.1) and Saibi (1999, §3.4.3).

One detail that bears notice is that the automatic definition of projections is keyed to record member names. Indeed, record projections are automatically defined *in the current scope* using names that are, in the corresponding inductive, given to arguments of the constructor- hence, morally, nothing more than bound variables used locally to resolve type dependence. On occasion, this may pollute the namespace.[22] Since there is plethora of mathematical structures defined using "a carrier equipped with some operations and propositions", it is clear that the developer of a library will have to come up for a flurry of distinct names for a projection to that carrier. But the overcrowding even starts with `P1` and `P2` above: propositions are rarely reused dependently inside the constructor, rarely used for programming generically, and what the projection's corresponding "rightful" names should be is unclear. It is true that they occur in proofs, but if needed, they can be retrieved by a simple lemma *unboxing* the inductive with case analysis, for example:

[22] Coq also automatically confers those projections special roles when they happen to be canonical members of `Structure`s. We will come back to this later (§ 1.3.1 on p. 51).

```
Lemma ordered_symmetric :
  ∀(T: orderedType), asym (carr T) (leq T).
Proof. case; trivial. Qed.
```

Fortunately, Coq will refuse to define a projection without an assigned name, *i.e.* one whose member name is the symbol _. Hence, it is customary to define only computationally meaningful projections (in `Type`), and to limit their scope to a given module, leading to a definition such as the one of Fig. 1.9 on the facing page. The corresponding members can

then be retrieved by, for example, aliasing qualified record names such as `OrderedType.rel` to `leq`.

Another difference between having native Σ-type definitions and the equivalent inductive definitions is that those records types cannot be defined dynamically — quickly whipped up in a proof, for example[23]. This is naturally a consequence of the encoding of records using inductive types, whose declaration adds to a signature separate from that of common terms.

We conclude with an example of a simple *instance* of our record, whose components come from the standard COQ library.

```
Module OrderedType.

Record class : Type := Pack {
  base : Type;
  rel  : base → base → Prop;
  _    : asym base rel;
  _    : trans base rel
}.

End OrderedType.
```

Figure 1.9: A record definition with limited scope, and parsimonious projection naming.

[23] While inductives can be defined within modules, on the other hand.

```
Require Import Arith.

Definition nat_ordered : OrderedType.class :=
  OrderedType.Pack nat le le_antisym le_trans.
```

1.1.8 *Dependent records, sharing, and inheritance*

The statement we made in § 1.1.1 on p. 19 is now clear : **COQ's dependent record types provide - at least in all non-propositional cases - the exact equivalent of telescopes, *i.e.* of iterated Σ-types.**

Now comes the question of how best to compose them to form mathematical contexts.

As we have mentioned before (on p. 18), "generalizations" at large have three well-known mechanisms: (a) the *specification* of a component containing the primary elements on which we will reason, (b) the *abstraction* of programs and proofs in the terms defined by that component, (c) and *instantiations* explaining how the concrete data on which we want to operate, compute, or prove fits the specification.

When those reasoning phases express themselves through programming language mechanisms that purport to implement a form of genericity (whether based on modules, records, or objects), it has now become customary to use a specific vocabulary[24] to describe each operation. They altogether describe a *concept* "pattern". [25] How this translates to COQ's records is perhaps best shown through the classic *apples to apples* example of Garcia et al. (2006) that I comment here with keywords italicized.

[24] (Austern 1998; Stepanov and Lee 1995)

[25] Though the name goes back to the original C++ template library, the notion gained enough popularity recently to lead to a recent and much-publicized (but ultimately unsuccessful) proposal for their inclusion in the pending C++0X revision of the language.
Oliveira et al. (2010) coined the corresponding "pattern" denomination — meant in the object-oriented sense of the term, as opposed to a specialized language construct.

In Fig. 1.10, both `ordType` and `apple` are independent *concepts*, specified through records. The definitions of `a1` or `a2` *model* the concept, declaring how their type fits the required members. `pick` is an *algorithm* on the `ordType` concept, whose definition is noticeably independent from any `apple`-related notion, and finally, the call to `Check` on the last line returns the type `apple`, and features an *instantiation* of the concept this function requires as its second argument.

```
Record ordType (A:Type) : Type :=
  OrdType { cmpare : A → A → bool }.

Record apple : Type := Apple { weight : nat }.

Definition ordApple :=
  OrdType apple (fun x y => weight x ≤ weight
    y).

Implicit Type T:Type.
```

```
Definition pick T (measure : ordType T) :=
   fun (x1 x2:T) => if (cmpare _ measure x1 x2)
    then x2 else x1.

Definition a1 := Apple 3.

Definition a2 := Apple 5.

Check (pick _ ordApple a1 a2).
```

Figure 1.10: Apples to apples with records

This simplest of examples amounts to the application of a function defined on a record type, seen under a slightly colored perspective. It already covers the day-to-day occurrences of generalizations we manipulate, and its most remarkable characteristic is perhaps *retroactive modeling* : here, we were able to fit `apples` into the `ordType` shoehorn necessary to apply `pick`, an operation which included side-effects (the application of projection `weight`), but without ever having to plan for those steps during the definition of `apples`.

Indeed, a common misconception is to think of the *concept* pattern as akin to the *interfaces* of object-oriented languages. But those languages would prevent us from defining a `pick` method on an `ordType` interface, in isolation, and, next, using it on instances of class `apple`. We would have needed to state, at the declaration of `apple`, that it *implements* the interface `ordType`.

More occasionally, though, we encounter concepts which do not fit in a model-to-concept relationship, in which any instance of one models the other, but feature either:

(i) a concept-to-concept relationship, in which the definition of one of the concepts has prerequisites (operations, properties) that include those of the other. We then say the former *refines* the latter.[26]

(ii) or as distinct concepts, possibly with some shared parameters, both occurring in the definition of a third.

[26] Note that this is a relation between *specifications*, akin to object inheritance. Its consequence for *models* is that the *refining* concept models the *refined* concept through the application of a "forgetful functor". See also § 1.2.7 on p. 45.

In both cases, we expect the specification language to provide us with a precise way to organize the *sharing* of the involved structures. In COQ, however, there is more than one way to achieve this. In the next four subsections we provide a description of the program hierarchization paradigms encountered in COQ formalizations. We put the emphasis on their structure taken in isolation, postponing the assessment of their ease of use until § 1.3 on p. 51.

Indeed, those paradigms interact in non-trivial ways with at least **four** programming facilities of COQ we will touch on later (implicit arguments resolution, notations, implicit coercions, and type inference, see § 1.2 on p. 37), and whose net effect is to make symbols **disappear** from user interaction, in distinct but related ways. We hope this fully-explicit description will make the later critical discussion of those structural mechanisms less confusing.

1.1.9 Pebble-*style sharing in Coq*

The most common way of expressing sharing or inheritance between concepts is to expose the points of synchronization as (universally-quantified) parameters of the structure. The target concept is then specified as a long dependent product, with every pre-requisite concept strung along in its parameters. This method, introduced by Burstall in the language *Pebble*,[27] is widely known as *pebble-style* sharing, and demonstrated in Fig. 1.11 on the next page.

[27] (Burstall 1984), though Pollack (2002) points to a more complex attribution.

This figure introduces a running example, in which the astute reader will remark a customary use of the equality and order functions of type

`(A → A → bool)` as if they were properties `(A → A → Prop)` (in the property of the `latticeType` structure). The goal of the use of booleans is to allow the simple implementation of an order for generic lists in § 1.1.12 on p. 34. It blends seamlessly with the use of those booleans as properties, thanks to a coercion `(is_true : bool → Prop)`, as explained in § 2.1.1 on p. 84. The details of this technicality are inconsequential here, since what we really want to focus on is architecture more than computation. We thus invite the reader to focus on the fact that in this first snippet, `latticeType` takes a parametric argument of type `eqType`, in characteristic Pebble-style fashion.

```
Record eqType A : Type := EqType {
  eq : A → A → bool
}.

Record latticeType A (eA:eqType A) :Type :=
  LatticeType {
    bottom : A;
    top    : A;
    meet   : A → A → A;
    join   : A → A → A;
    lt     : A → A → bool;
    _      : ∀x y,
      (lt x y) ⟷ (eq A eA (join x y) y)
  }.
```

Figure 1.11: A lattice Type, defined using pebble-style. The early work of Jones (1992) in the framework of Haskell's type classes inspired this example. Contrarily to what I did on p. 27, I chose to keep the programmative flavor of the boolean equality predicate to stay close to the original example (implying the decidability of that relation, see § 2.1.1 on p. 84) and to simplify the later definition of a generic list order. We use the fact that booleans coerce to properties as we will explain in detail in § 2.1.1 on p. 84. The only practical consequence of this choice to the matter at hand is that the custom equality allows us to see how second-order inheritance works: the lattice depends on order which depends on equality.

It is folklore that this style of sharing "does not scale" (Pollack 2002; Sacerdoti Coen and Tassi 2008), but while I agree, I would like to argue this more precisely in the next few sections: this style of sharing allows for maximal flexibility because it avoids unnecessary levels of boxing, allowing the user to decide at modelling time which elements to bundle to form a concept instance. Gonthier leveraged this advantage to the extreme - that is, with records in which all relevant data is in parameters - when he manipulated paths of combinatorial graphs in a large-scale development (Gonthier 2005), for example. However, paths describe a concept where there is no one-to-one relationship between the bundled elements, namely case (ii) above (on the facing page).[28]

In the far more common case of inheritance (i), however, this style involves explicitly handling a string of parameters that quickly grows in size with the development of a well-tiered library.

For instance, to model our lattice of Fig. 1.11, we would have to pass two arguments: the carrier type, and an equality model on that type. But one can easily imagine wanting to extract the decidable total order relation that it contains (named `lt`), defining it separately in the style of an `ordType` such as in Fig. 1.10 on p. 29, and specifying the lattice based on that concept.

Better yet, in Coq, we require stronger verification concerns than Garcia et al. (2006), and expect an order relation to carry a more precise specification, similarly to the `orderedType` defined on on p. 27. We define this refined notion of lattice in Fig. 1.12 on the following page.

This thinner slicing of our concept therefore involves passing three arguments for every instance declaration of our lattice. Given our choice to specify equality and order as independent records, it seems unavoidable to

[28] See Garillot et al. (2009, §2.1) for details of the aforementioned structure.

```
Record eqType A : Type := EqType {
  eq : A → A → bool
}.

Definition asymm (T:Type)
  (eT:eqType T)
  (o: T → T → bool) : Prop :=
  ∀(x y : T), (o x y) →
    (o y x) → (eq T eT x y).

Definition transit (T:Type)
  (o: T → T → bool) : Prop :=
  ∀(x y z : T),
    (o x y) → (o y z) → (o x z).

Record orderedType A (eA: eqType A) : Type :=
  OrderedType {
    ord  : A → A → bool;
    anti : asymm A eA ord;
    tran : transit A ord
  }.

Record latticeType A (eA:eqType A)
  (oA : orderedType A eA):Type :=
  LatticeType {
    bottom : A;
    top    : A;
    meet   : A → A → A;
    join   : A → A → A;
    _      : ∀x y,
      (ord A eA oA x y) ↔
      (eq A eA (join x y) y)
  }.
```

Figure 1.12: A well-tiered pebble-style decidable lattice *concept*-ualization.

provide them a lattice model at one point or another. However, pebble-style sharing requires synchronizing the equality type argument passed to the order relation `oA`, with the one used in the lattice specification itself (`eA`). Given a sufficient level of concept nesting, this quickly becomes tedious.

Unfortunately, in mathematical developments, thin-slicing a concept hierarchy is often the best way to maximize the usability of the library. It indeeds allows the user to express his properties and lemmas at the maximum level of generality, roughly making him adhere to the "single responsibility" principle. Those frequent refinements are therefore a natural evolution of a development over time, while on the other hand, mathematical structures often include half a dozen operations or more. We will come back on programmatic tools developed in COQ to mitigate the heaviness of this style (§ 1.3.1 on p. 51), but for now, we turn to a more structural alternative.

A third option which is often mentioned when dealing with records in type theory: *manifest types*, which is an ascription mechanism that involves exposing equational relations on members of a record type. They are somewhat irrelevant to my work, in which the encoding of records as inductive types imposes certain constraints, but their expressiveness and computational performances deserve some note (see Pollack (2000, 2002) and Tassi (2008, §5.3) for a discussion and pointers to the literature). We will however treat an expressivity challenge to which manifest records would by design provide a good solution in § 1.4.3 on p. 70.

1.1.10 Telescopic sharing in Coq

Nested COQ records are not telescopes in the proper sense. However, if one chooses the discipline of embedding superclasses as record members, it is easy to see why we might call that *telescopic*: if telescopes are the iteration of nested dependent pairs, here, we simply try to manipulate concepts as nested dependent tuples, *i.e.* nested dependent records. We provide an example of the lattice defined in the last section, translated to telescopic style, in Fig. 1.13 on the next page.

This method of nesting records has the advantage of limiting manipulations of the 'superclasses' of a given type. To model how a lattice stems from an order relation, you provide the order relation once, and access the corresponding equality through record projections. The downside is that the manipulation of iterated projections can be verbose, and proportional to the depth of *concept nesting*. We will see later what programmatic tools exist in COQ to handle the overhead of such manipulation (§ 1.2.7 on p. 45).

Nevertheless, the important point is that this solves, or at least moves, the problem created by the proliferation of arguments in pebble-style sharing: the heaviness has shifted from the modelling time to the specification time, moving thus from the user to the developer of the library, and from the frequent case to the less frequent.

```
Record eqType : Type := EqType {
  carr : Type;
  eq   : carr → carr → bool
}.

Definition asymm (eT:eqType)
  (o: carr eT → carr eT → bool) : Prop :=
  ∀(x y : carr eT), (o x y) →
    (o y x) → (eq eT x y).

Definition transit (T:Type)
  (o: T → T → bool) : Prop :=
  ∀(x y z : T),
    (o x y) → (o y z) → (o x z).

Record orderedType : Type :=
  OrderedType {
    sort : eqType;
    ord  : carr sort → carr sort → bool;
    anti : asymm sort ord;
    tran : transit (carr sort) ord
  }.

Record latticeType : Type :=
  LatticeType {
    osort  : orderedType;
    bottom : (carr (sort osort));
    top    : (carr (sort osort));
    meet   : (carr (sort osort)) →
             (carr (sort osort)) →
             (carr (sort osort));
    join   : (carr (sort osort)) →
             (carr (sort osort)) →
             (carr (sort osort));
    _      : ∀x y,
      (ord osort x y) ↔
      (eq (sort osort) (join x y) y)
  }.
```

Figure 1.13: A well-tiered telescopic-style decidable lattice *concept*-ualization.

It is therefore unsurprising that this method has known some popularity in formalizations of abstract algebra (Geuvers 2002; Jackson 1994, 1995; Pollack 2000; Rudnicki 2001), where hierarchy depth quickly becomes an issue.

A point of note, however, is that in this nesting discipline, one may quantify on a given concept, but that dependent product can no longer refer to superclasses of the concept. While this is manageable for the vertical extension of a concept, quantifying on concepts that share some parameters becomes more difficult, as noted by Sozeau and Oury (2008, §4.1). They point to the example of a (categorical) adjunction,[29] composed from two "crossed" functors, whose source and end category must correspond. Only by having those categories as parameters of the functor concept can the type of an adjunction guarantee that condition by itself. With the strict discipline of having superclasses as members imposed by telescopic-style, there is no way to check the chiasmatic constraint on the source and target categories of functors at a *type level*. We can only compare the *values* of projections of our models.

[29] We will come back to the example of the adjunction, suggesting a better solution, in § 1.4.3 on p. 70.

The loss of expressiveness, though genuine, is however reversible. As we have already noted on on p. 29, concepts make the translation between specifications easy enough to support switching as needed between telescopic and pebble-style sharing, when more expressiveness becomes necessary.

1.1.11 *Pebble inefficiencies*

Let us now look at a boolean lattice model declaration, assuming we have specified the lattice concept using *pebble*-style sharing. I use notations for boolean functions defined in the `ssrbool` library, and omit proofs, whose content hardly matter here :

```
Definition bool_eqType :=
  EqType bool (fun x y => x == y).

Variable ordbool : bool → bool → bool.

Definition bool_orderedType :=
  OrderedType bool bool_eqType ordbool antiH tranH.
```

```
Definition bool_latticeType :=
  LatticeType bool bool_eqType bool_orderedType
            false true orb andb coherentLatticeP.
```

Let us make the assumption that there is no underlying sharing in the way COQ treats sub-terms.[30]There are at least[31] two occurrences of `eqb` (a.k.a. `fun x y => x == y`) in `bool_latticeType`:

- one occurs in the `bool_eqType` parameter,
- one occurs after δ-expansion in the `bool_orderedType` parameter,

In fact, if we continue specializing the `latticeType` concept, one can see how the number of occurrences of `eqb` grows quickly. Indeed, at the k-th level of subtyping, an instance of our k-times extended structure passes its $(k-1)$ parent instances as arguments to the constructor. It thus contains the sum of all occurrences of the first-level member `eqb`. If we call M_k the number of members of a record at the k-th level, and if we bound the number of members we add at each subtyping operation by a constant C:

$$M_{k+1} = \sum_{i=0}^{k} M_i + C = 2M_k$$

Naturally, **this entails that Pebble-style provokes an exponential size increase in the size of the term relative to the subtyping depth**.

Let us now look at the same declaration for a boolean lattice, written in telescopic-style (`bool_eqType`, `ordbool` stay unchanged):

```
Definition bool_orderedType :=
  OrderedType bool_eqType ordbool antiH transitH.

Definition bool_latticetype :=
  LatticeType bool_orderedType false true
              orb andb coherentLatticeP.
```

Telescopic style solves the problem in term growth with refinement by collapsing the dependency of first and second (*etc*) order subtypes (order & lattice respectively) into a single reference. But was all that necessary ? Surely, since even the largest hierarchies have a subtyping depth of about 10 or so (Garillot et al. 2009, §3)[32] modern provers can manipulate a limited number of instances whose base size (moderate in practice) grew only by a few orders of magnitude ?

We give a conflicted answer: since *interactive* tactics in COQ are already frequently non-linear in the size of the term they manipulate, one might fear a threshold at which the computational complexity of working on a term will start to frustrate users. We will treat a specific and frequent example where COQ needs to compute on head normal forms of records in § 1.4.4 on p. 74. But a distinct problem, discovered using the same basic insight — namely that dependent *parameters* without sharing in term representations are a burden — compounds this concern.

1.1.12 Telescopic inefficiencies

Mathematicians like to manipulate uncomplicated nested objects, such as polynomials of matrices, keeping the associated nesting of structural properties implicit: assuming those matrices have vanilla coefficients belonging to

[30] This assumption is safe. The reliance of COQ on a Hindley-Milner style type inference engine means that parametric arguments have to be accessible in full inside a term's external representation. At the internal level, one can imagine representing term ASTs as DAGs with full sharing, but this is not the model adopted by the implementation of COQ, which performs eager ζ-reduction before convertibility checks.
Finally, kernel-level procedures do attempt to act on terms modulo congruence closure using hash-consing, but are at the moment subject to a (reported) bug that makes them inefficient in that respect.

[31] This minimal account assumes that `eqb` is not used in the definition of any other computational member of a subtype, such as `ordbool`, or in any of their proof terms, such as `antiH` or `coherentLatticeP`. This is a risky bet, even in this limited example.

[32] ... and since the average user needs much less: the depth of the closest type-class-based libraries we know (Geuvers 2002; Odersky and Moors 2009) stays under the half-dozen mark, and yet those are still considered exceptionally large.

an arbitrary ring, a three-level-deep nesting of rings. This occurs frequently — in fact as soon as one introduces structures that are *containers* in the moral sense of the term, i.e. objects like trees or vectors, which are commonly represented as a structured way to carry a generic payload of data. Experience also shows this comes up in fundamental theorems (Bertot et al. 2008, §6). Though I do not want to delve into the specifics of the SSReflect library just yet, we can have a look at this nesting of concepts by emulating it with lists.

Let us specify the lexicographic order on lists, and the corresponding `orderedType`:[33]

```
Definition eql (eT: eqType) (l1 l2: list (carr eT)) :=
  if length l1 == length l2 then
    reduce andb (zip_with (eq eT) l1 l2) true else false.

Definition list_eqType (eT:eqType) :=
  EqType (list (carr eT)) (eql eT).

Definition ordl (oT:orderedType) (l1 l2:list (carr (sort oT))) :=
  let m :=
    (filter
      (fun c =>
         let: (x,y):= c in
           ~~ eq (sort oT) x y)
      (zip_with (fun x y => (x,y)) l1 l2)) in
   if m is x :: xs then
     let: (a,b) := x in
       ((ord oT) a b)
   else
     (length l1 ≤ length l2).

Definition list_orderedType (oT:orderedType):=
  OrderedType (list_eqType (sort oT)) (ordl oT)
              (ord_antiH oT) (ord_transitH oT).
```

If we now look at lexicographic order for lists of lists of booleans (`bool_list_list_orderedType`, Fig. 1.14), we have a good idea of what the three-level ring of the Cayley-Hamilton theorem looks like at the structural level. A mathematician uses it to compute by wielding the addition of his ring. The equivalent for us is the order's comparison function, namely `(ord bool_list_list_orderedType)`:

it δ-expands to:

```
ord (OrderedType
      (list_eqType (sort bool_list_orderedType))
      (ordl bool_list_orderedType)
      ...
     )
```

but each of the arguments of the constructor, say `ordl bool_list_orderedType`, itself δ-expands to:

```
ordl (OrderedType
        (list_eqType (sort bool_eqType))
        (ordl ordbool)
        ...
      )
```

In other terms, nested records definitions, in telescopic style, regroup members, every one of which is a function application parametric in the

[33] In order to show runnable code with a minimal number of dependencies, we switch back to `list` in our examples, rather than its `seq` re-implementation provided in the Mathematical Components libraries. Please consider those two aliases for the same notion, for the purpose of those examples.
For brevity I use the (hopefully intuitive) SSReflect pattern-matching syntax (Gonthier et al. 2008, §3.), and assume the well-known higher-order functions:

- `zip_with` (a.k.a. `map2`), a dyadic `map`,
- `filter` from `Lists` in the COQ standard library,
- `reduce` (a.k.a. `fold`), the standard list catamorphism,

Exceptionally, I omit implicit type arguments of those three functions, and use boolean equality and inequality similar to those defined on integers in the `ssrnat` library. As usual, I omit proofs of `antiH`, `transitH`.

```
Definition bool_list_orderedType :=
  list_orderedType (bool_orderedType).

Definition bool_list_list_orderedType
      :=
  list_orderedType
     (bool_list_orderedType).
```

Figure 1.14: Nested lexicographic order instances for nested lists of booleans

most immediate "nestee". Indeed, we have defined `eql` (resp. `ordl`) independently and generically in term of some arbitrary `eqType` (resp. `orderedType`). Those functions encapsulate each one full copy of the "nestee", which, if there are C such members, multiplies the size of the term by that many.

Since this occurs at every nesting level, we can conclude that, in telescopic-style, term size grows in the order of C^n with the nesting depth of any given subterm. In the case of the Cayley-Hamilton theorem (Bertot et al. 2008), where $n = 3$ and $C = 15$ (a ring structure of 15 members nested thrice), we quickly created objects on which we could no longer compute diligently. In general, with a system that supports certain facilities (implicit arguments and coercions), the user could (and did) go on specifying for a while before realizing that concepts of her library led to unmanageable terms.

The term bloat in fact occurs for wanting to give a monolithic specification to each nesting construct. When we built a lexicographic order on lists above, we defined generic functions that took as argument just the correct level of structure they needed and no more: to define a decidable equality on lists of elements with `eql`, you need a decidable equality predicate for those elements, but you do not care whether you can order them yet.

And yet, at definition time, we pass `eql` a full copy of the nesting record, immediately projecting it to its first member. Without early ι-reduction under the inductive instance constructor, the duplication is painfully obvious.

We leave it to the reader to adapt the example of Fig. 1.14 on the previous page in pebble-style. He will notice that, generic nesting functions (analogous to `list_eqType` and `list_ordType` above) take a *nestee* as argument according to a pattern exactly similar to the *inheritance* of pebble-style records for superclasses. This means that the growth in term size, when nesting pebble-style records, is also exponential. In this example, we measure that once appropriately δ-expanded, `bool_list_list_orderedType` is about twice as long a term in pebble-style as it is in telescopic style. We estimate this accounts for a more parsimonious structuring of refinement when going from equality type to ordered type in the telescopic case. As we will see later (§ 1.4.1 on p. 60), this difference becomes negligible as the nesting depth augments.

In summary, the fact that record introduction and elimination forms package a copy of their parameters — when they are already present in the term anyway — incurs a term growth that kills pebble-style inheritance, and nesting of both telescopic and pebble style. Brady et al. (2004) observed that storing a copy of these arguments in introduction forms was unnecessary, and we think notice should be given to a recent development on their necessity in elimination form (see note 83 on p. 80).

1.2 *From user input to typed term: a reappearing act*

FROM USER INPUT TO THE FINAL INTERPRETATION COQ MAKES OF A TERM, A NUMBER OF DISTINCT STEPS OCCUR. All allow the user to specify less information at user input, and there is thus to gain by a precise understanding of what they entail.

The first three feature in an example in Fig. 1.15. Note `ex_intro` comes from the definition of `ex` in § 1.1.6 on p. 26.

1.2.1 *Pre-inference*

The first transformation that occurs to user input in COQ is the expansion of notations (Coq 2010, §12). An important detail to note is that this occurs in an untyped fashion.

Coq then processes the meta-expanded term to provide the appropriate number of arguments to each λ-abstraction. User definitions can indeed feature an incomplete set of arguments in applications, on which the system will have to elaborate. This *resolution of implicit positions* was originally a mechanism featuring purely syntactic sugar, implemented by Saibi (1999, §4.6) as a particular flavor of η-expansion: when inspecting the type of a definition, COQ assumes variables occurring free under the context of the previous (provided) arguments are in fact inferrable (missing) arguments that it must synthesize.

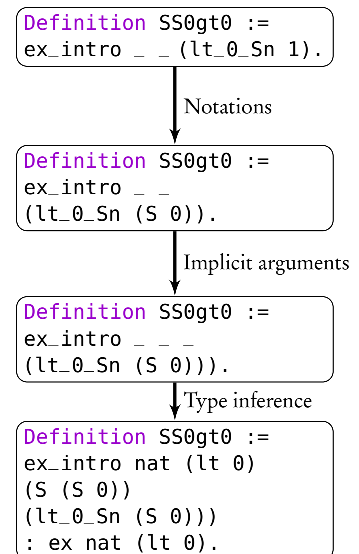


Figure 1.15: An example of term elaboration, from notations to type inference.

The main goal of the mechanism is thus to provide placeholders (marked with _) to indicate arguments that a simple unification algorithm can infer from the declared type of a λ-expression, or from the type of other arguments. Since then, the user has gained finer control over how and which arguments the system should elaborate from a definition (Coq 2010, §2.7). However, we note one constant, namely that implicit argument resolution occurs before type inference, and operates entirely at the syntactic level. This design choice owes to the approach of Pollack (1990) which consists in having an *implicit syntax*, which allows conservative extension upon the *explicit syntax*, to which implicit terms translate before type checking.[34]

This implicit argument mechanism has become the point of entry of `Class` *constraint propagation*.[35] We refer the reader to Sozeau (2008, §7.1.2) for more details. We will come back on the equivalent mechanism for `Canonical Structure`s later (§ 1.3.2 on p. 52).

[34] This approach will also influence Saibi's design as described in § 1.2.7 on p. 45.

[35] The *instantiation* phase occurs in a second phase, as the last bit of type inference. Actually, the term is well typed even if `Class`es are not resolved at all.

1.2.2 *Type Inference*

A unification problem COQ allows the user to avoid entering explicit type annotations, and tries to guess the type information from the information present in explicit terms. This type inference mechanism relies on unification: the gist of the mechanism consists in assigning each untyped product a variable type; then considering constraints between those variables using the typing rules of the calculus.[36] Those constraints elaborate to equalities between the types of user-input terms (which, in our calculus, are of course terms themselves). Hence, the unification problem consists in, given

[36] Reducing Hindley-Milner-like type inference to unification is a process embedded deep enough in the fabric of the ML-inspired family of languages that we are content with this informal hint. Pottier and Rémy (2005) provides a comprehensive reference on how it happens in the first-order case. Its principles carry over well for the higher-order case. (Saibi 1999, §4.4) gives the gory details of constraint generation for COQ.

two typed λ-terms e_1 and e_2, finding a substitution σ for the free variables of those terms such that $\sigma(e_1)$ and $\sigma(e_2)$ are convertible. We call such a substitution σ a unifier of e_1, e_2.

In practice, we will see that we quickly come to consider an extended version that problem on e_1, e_2 to the case of a vector of pairs of terms $\mathbf{e_1}, \mathbf{e_2}$ to be solved for simultaneously, for which we will call a substitution σ an *unifier* of the vector when it is an unifier of each pair $e_{1,i}, e_{2,i}$ for $1 \leq i \leq n$. Those pairs are often called *disagreement pairs*.

In COQ, the unification algorithm plays a fundamental role for type inference, but also during rewriting, for instance. Our tightly-focused discussion will remain silent on a number of its other applications. The interested reader can consult landmark survey articles for the first-order case (Knight 1989), and for the higher-order case (Dowek 2001; Huet 2002).

The first element that deserves notice is that a unification problem posed by type inference based on the input of a COQ user is the *order* of the problem. This order is the functional level of the metavariables found in disagreement pairs. When doing type inference for the simply-typed λ-calculus (*à la Church*), we are only looking to assign type variables, so that first-order unification suffices. Since COQ types have two flavors of binding structure, namely polymorphic and dependent types, there are two occasions for type inference to present *higher-order unification problems*. An example is given in Fig. 1.15 on the preceding page, with inferring the second argument to `ex_intro`, namely `(lt 0: nat → Prop)`.

Another fundamental definition used in this approach to type inference is that of a *most general unifier*. A unifier σ of a problem is the smallest, or the most general, if for all unifiers ω, there exists a substitution σ' such that $\omega = \sigma' \circ \sigma$. Then we say that ω is an *instance* of σ. In practice that means that σ is no more specific than ω, and that it suffices to rename some variables of the output of σ to get that of ω.

For a language with only limited forms of polymorphism (such as ML, for example), having a first-order unification algorithm returning the most general unifier means that type inference returns a *principal type*: a type which constricts the term as loosely as possible while still ensuring consistency with the type system (and the sought-after progress and safety properties). When the unification algorithm does not know how to compute a most general unifier — and in fact when the unicity of the smallest unifier is not guaranteed — this means that the output of unification is at best non-deterministic or incomplete.

Higher-order unification In fact, if we consider the following higher-order unification problem, where f is a (e.g. first order) functional constant and ?M a meta-variable for which we want to solve:

$$\lambda x\,.\,?\mathrm{M}(f\,x) \triangleq \lambda x\,.\,f(?\mathrm{M}\,x) \tag{1.1}$$

Then any substitution σ_k for $k \geq 0$, such that:

$$\sigma_k = [\mathrm{M} \mapsto \lambda x\,.\,f^k\,x]$$

is a unifier for (1.1), and none of those unifiers is an instance of another. In fact, the set $\{\sigma_k \mid k \geq 0\}$ is even a *complete set of unifiers*, in the sense that any other unifier for (1.1) is an instance of an element of that set. It is however remarkable to notice that the complete set of unifiers of that problem is infinite.

The well-known consequence is that type systems with an unlimited binding structure in types cannot hope for complete and terminating type

inference.[37] In fact, higher-order (and even second-order) unification are undecidable, though this is not the result we want to draw attention on given our focus on inference.[38] We focus instead on what can be achieved in practice to obtain a reasonably complete but *strongly reproducible* process to infer types in COQ.

[37] J. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1-3): 111–156, June 1999. ISSN 01680072. doi:10.1016/S0168-0072(98)00047-5

[38] The interested reader will find all references on the decidability of higher-order unification problems in the bibliography of the survey articles mentioned on the preceding page.

Huet (1975) published a semi-decidable algorithm to solve higher-order unification in Church type theory. It is in fact a pre-unification algorithm that works on β-normal forms of the calculus. One of its key insights is to separate disagreement pairs in categories of equations, grouping separate terms. Let us look at a β-normal term:

"The non-determinism, more so than the undecidability, presents some problems with full higher-order unification as the basis for proof development environments and logic programming languages."
(Pfenning 1991)

$$\lambda \mathbf{x} \,.\, u\,\mathbf{t} \tag{1.2}$$

u is the *head* of the term. A term in a disagreement pair is *rigid* when the head of its normal form (u) is not a meta-variable, and *flexible* otherwise. Huet noticed that disagreement pairs between flexible pairs were always solvable. He concluded that it sufficed to provide substitutions that solved the *other* disagreement pairs to leave the problem in a form that any heuristic with a uniform behavior could finish off. This discipline of solving higher-order unification up to flexible-flexible pairs is since then called *pre-unification*.

The pre-unification algorithm of Huet was extended to calculi with dependent types (Elliott 1990; Pym 1990), and those extensions retain the same approach of substituting up to flexible-flexible pairs. However, for a calculus with dependent types, the assumption that flexible-flexible pairs always have solutions is no longer valid. For example, if ?X is a meta-variable with type $\lambda x : \texttt{Type} \,.\, x$, the following problem:

$$?\mathrm{X}\,(\mathrm{A} \to \mathrm{B})\,a = ?\mathrm{X}\;\mathrm{B}$$

has no solution (Dowek 2001). The semi-decidability of the algorithm for higher-order unification obtained by Huet for simply typed theory does not carry well to more complex calculi.

Hence, hoping to devise a general and complete algorithm to solve type inference seems particularly intractable in the context of COQ's meta-theory. Contrarily to systems like λ-PROLOG or ISABELLE, for example, COQ takes the practical approach to dealing with this inherent incompleteness by not implementing Huet's algorithm directly. We therefore detail the parallel evolutions of the meta-theory with the practical approach of the implementation.

1.2.3 *Unification in* Coq

Well-founded tricks In some of the next few sections, we are going to expose and demonstrate manipulations of the unification procedure in COQ.[39] Those implementations are based on the interplay between `Canonical Structure` instance resolution and specific properties of the unification procedure: the syntactic head constant matching on weak-head normal terms, the late δ-reduction, and the timing of coercion insertion.

[39] To be crude about it, we are going to explain how to obtain subtyping from `Canonical` instances corresponding to supertype injections (§ 1.4.1), how to program deterministic overlapping instance selection for `Structures` (§ 1.4.4), and how to implement `Structure` projection insertion through notational phantom type setups (§ 2.3.2).

The key point for us is that, if COQ's history is any indication, the unification procedure is inherently volatile. Nonetheless, our manipulations

are valuable because the crucial characteristics we rely on are defensible as useful and sensible improvements of higher-order unification. Rather than expose the unification procedure in detail, we thus strive to place those key features on the more solid foundation of the literature on unification in COQ.

COQ literature versus its implementation The implementation of COQ (started in 1984) is, since the founding work of Coquand and Huet (1989), a *syntax-directed* version of, roughly, a PTS extended as described in § 1.1. Saying the calculus is *syntax-directed* means in particular that the engine solves the problem of type-checking by *type synthesis*, through unambiguous applications of the typing rules of the calculus. It inspects the shape of a term of the calculus (in a suitably reduced form) knowing that if this term can be well-typed, it will match the conclusion of exactly one of the rules of the calculus. It suffices to repeat this work on the hypotheses of that rule, recursively, to deduce a finite type derivation for that well-typed term. Untypeable terms going through the same process eventually fail to match the conclusion of any typing rule.[40]

The distinction matters since, in *all* standard presentations (Coq 2010, §4) — as well as in ours — the calculus of COQ is *declarative*, meaning in particular that one cannot use rule CONVERSION (Fig. 1.2 on p. 21) unambiguously to produce such a type derivation. Thankfully, because the calculus of constructions is *semi-full*, it is a standard result that the syntax-directed presentation is equivalent to the declarative one (Pollack 1992).

Unfortunately, most of the literature about the meta-theory of the calculus of COQ operates on a declarative presentation of the calculus. Moreover, beyond this discrepancy, the literature of the CIC does not include η-expansion in the conversion relation,[41] because of its difficult interaction with the introduction of the universe hierarchy. Both gaps are active research topics to this day (Siles 2010).

By and of itself, the absence of the η-conversion rule in COQ complicates the unification algorithms. Huet (1975, §4.5), who took the precaution of formulating his algorithm for a simply-typed λ-calculus with no η-conversion, details the simplifications that can occur when this rule, equivalent to functional extensionality, is added to the calculus.

[40] A particular consequence of this choice of semantics is that the type inference algorithm can (and does) analyze the applicative structure of terms and compare them to the conclusions of typing judgements by converting them to *weak head normal form*, a form obtained by applying β-reduction "only at the head" of the term, in the sense of equation (1.2). We will see how this bears some consequence in the implementation of unification for COQ.

[41] With the notable exception of (Altenkirch 1993) and Werner (1994) which predate the removal of η-expansion. Until late September 2010, even the development version of COQ did not include this reduction rule either. The η-expansion has been re-introduced in COQ's trunk, and will be part of the next release (following 8.3).

Unification by Transformations The formal study of unification in literature (Dowek 1993; Elliott 1990; Saidi 1994) leading up to the description of a higher-order unification algorithm for CIC (Cornes 1997) has historically adopted a slightly different presentation from the one of Huet (1975). It relies in fact on a presentation of unification *by transformations*. This approach was used and exposed in detail in the work of Snyder and Gallier (1989), in an introductory paper that generalized Huet's approach to higher-order unification, while still dealing only with simply-typed λ-calculus. If we consider a set of disagreement pairs as a *disagreement set* $\mathcal{S} = \{(\mathbf{u}, \mathbf{v})\}$, the method consists in applying simple transformations, including substitutions akin to variable elimination, until the obtention of a solved system $\mathcal{S}'$ whose solution is clear.

In all cases where this method has been extended to richer and richer

COQ does not strictly use unification by transformations, in the sense that the order in which it treats disagreement pairs is not prompted by reasoning on flexibility (it treats pairs in a strict left-to-right, depth-first order, see Gonthier et al. 2011, appendix). But since various unification heuristics fire on a pair in a manner similar to the transformations — in a fashion we are going to expand on in the next few paragraphs — it seems that this is roughly the model of COQ's unification one should have in mind.

calculi (mentioned above), this means closed flexible-flexible pairs. We have already seen that this *pre-unification* solved form does not necessarily entail a unification solution for the polymorphic, dependent calculus we care about, but a more pertinent consequence is that this is a *non-deterministic* set of abstract operations for unification, of which we can think of as a set of *inference rules*. As spotted by Snyder and Gallier,

> This removal of control and data structure specification allows us to examine the fundamental properties of the problem more clearly.

Hence, when Saibi (1999, §4.3) uses that presentation based on transformations to depict his implementation[42] of unification in COQ, he glosses over of the interplay between the transformations he uses. This might seem ill-advised since the description of Saibi is entirely practical and geared towards an implementation (he does not reason on his algorithm, and acknowledges it is incomplete). Moreover, this loosely-specified control flow reflects the actual implementation in COQ to this day: the unification routines defer to each other by continuation in an order which evolves along successive COQ versions. However, this approach gives a simple and suitable context for extending the algorithm with the application of specialized transformations when encountering terms of a given distinguished form. We will mention such an extension in the next subsection.

[42] The basis for the modern implementation of higher-order unification in COQ is a re-implementation by Amokrane Saibi and Christine Paulin. It used as a "first draft" that of Chetan Murthy, for which no documentation exists. The implementation went through numerous evolutions since, on which we will comment in the next paragraph.

Practical heuristics The implementation of higher-order unification in COQ does not try to achieve an enumeration of a complete set of unifiers. Indeed, in a process such as type inference, a partial solution giving good practical results seems sensible: it is always possible, at worst, to ask the user to fill type annotations.

The basis for the treatment of a higher-order unification problem set by type inference, in COQ, is thus a set of transformations devised by Saibi (1999, §4.3) and Paulin-Mohring.[43]

[43] *"Nous adoptons pour notre part un algorithme* ad hoc*, non complet mais avec un bon comportement en pratique. [...] Nous ne prouvons aucune propriété de l'algorithme donné ; nous nous contentons de spécifier la propriété de correction souhaitée."* (Saibi 1999)

It features the classic core insights of a higher-order unification algorithm (simplify rigid-rigid equations, construct solutions to flexible-rigid solutions using elementary substitutions) and of its adaptation to peculiarities of the calculus of COQ— a *scission* heuristic, *accounting equations*: see Dowek (2001). However, its elementary substitution is heavily skewed towards the *imitation* rule of the classic algorithm (Huet 1975; Snyder and Gallier 1989, resp. §3.4.1.1 and §4.9), and the additional difficulty of uniformly providing solutions to flexible-flexible pairs ensures it waives any claim to completeness.

But before trying to apply those rules, COQ attempts to use other heuristics, among which are, in this order, $\beta\iota\zeta$-weak-head reduction, first-order unification, and Miller-Pfenning pattern unification. The latter is a procedure for higher-order unification that applies on a special form of disagreement pair,[44] on which unification is not only decidable but linear (Miller 1991; Pfenning 1991), and whose implementation in COQ was introduced by Hugo Herbelin in version 8.1 γ. This progressive application of heuristics ensures the responsiveness of the proof assistant during the interactive process of type inference.

[44] A term is a higher-order pattern if each meta-variable has distinct bound variables as arguments. For instance, $\lambda x, y\ .\ ?F(x, y)$ is a higher-order pattern, but $\lambda x\ .\ ?G(?H(x))$ is not.

1.2.4 Implementation Highlights

As we have argued, there are differences between the approaches to unification taken by COQ and the literature on the Interactive Calculus of Constructions (and, to a degree, the literature on higher-order unification in general):

- COQ has a practical approach leveraging heuristics, and does not aim to be complete
- the calculus reflected by the implementation is syntax-directed, which skews it towards working with *weak head normal forms*,
- and it has no η-conversion

By itself, the syntax-directed flavor of the implementation is not a strong theoretical problem: the Church-Rosser and strong normalization properties of the calculus (Werner 1994, §4,5) allow us to replace convertibility by reduction to a common normal form. This makes reasoning on declarative presentations of the calculus valid with respect to conversion in our context, up to minor rearrangements.

Moreover, some of the particularities of the implementation find echoes and justification in the literature. For instance, approaching the implementation under the angle of transformations gives some weight to the work of Elliott (1990, §4.2) on $\lambda\Pi$,[45] which also shares that trend. There, he calls the simplification of term pairs by application of transformation rules *decomposition*.

[45] The specifics of that calculus appeared in § 1.1.3 on p. 21.

In particular, he goes on to show that it is not necessary, for $\lambda\Pi$ calculus, to reduce a pair of terms to the common full head-normal form before decomposing it: while *head-normal* form requires doing β-reduction even inside an abstraction, this is not needed to know which transformation rule one should apply. Weak head normalization — in which reduction at any inner level is omitted — suffices. The proof relies on the exact way convertibility checks happen in $\lambda\Pi$: convertibility is tested by reducing expressions to weak-head normal form and proceeding recursively on subexpressions of these normal forms (Coquand 1996). Since this is also true for the calculus of constructions (Huet 1989), the intuition is that it also suffices for COQ. Hence, **the optimization that consists in implementing unification as transformations on weak head normal terms is well-justified**: it is a sensible reuse of existing primitives, and a computational simplification, for all calculi that feature the same syntax-directed presentation obtained from type-checking *à la Pollack* (Pollack 1992).

Another example of a design choice in implementation that finds echo in the COQ-related literature is a heuristic adopted by Cornes (1997, §4.2). Contrarily to previous publications (Paulin-Mohring 1993; Werner 1994), the implementation of inductives in COQ never represented them as simple λ-abstractions (*i.e.*, their constructors). It was made in a more efficient manner, by storing them in a context, creating a discrepancy with literature that the habilitation thesis of Paulin-Mohring (1996) closed soon after. In this concrete context, especially given that the unification algorithm does not create new inductives, it is clear that it is advantageous to minimize "inductive accesses", i.e., the application of parameters to inductive types, which unfolds their definition through reduction.

In the implementation, this is done by singling out the unfolding of inductive types within a special form of reduction, δ-reduction, and preventing the δ-reduction of closed terms in disagreement pairs. The treatment of Cornes (1997, §6) shows that, for a declarative CIC with $\beta\eta$-reduction, this optimization preserves the completeness of the pre-unification algorithm. **This provides us with a strong indication that delaying δ-reduction is a sensible optimization of unification in the presence of inductive types**[46]

[46] We will reuse that fact in § 1.4.1 on p. 60 and § 1.4.4 on p. 74.

We will later come back on the importance of those two particularities of the unification process.

1.2.5 Extending Type inference with record heuristics

Let us suppose we have a mathematical structure, such as a type equipped with an equality function, defined in pebble-style, as in Fig. 1.13 on p. 33. In effect, the projection `carr` is a "forgetful functor" on type `eqType`:[47] it takes any term which happens to be an instance of this record type, say `bR : bool_eqType` (p. 34), and erases the equality function from its representation, leaving only the first member, here an element of type `bool`.

[47] We would ask some leniency on the part of the reader for this expression: except in one obvious occasion, we are going to use it to refer to erasure rather than an explicit categorical structure.

In mathematical practice, we often ask readers to remember, rather than to forget. How to compute the equality of two multisets, for example, is considered obvious, even though the syntactic notation commonly adopted for multisets — which often does not put multiplicities in any particular evidence — does not give the details of the eventual definition. In another instance, blackboard proofs in algebra often combine the groups they mentions using a number of set-theoretic operations with no additional justification, since how the group properties carry over set-theoretic operations was "treated in class".

A specific heuristic in the unification process of COQ purports to do exactly that: *inverting record projections*, by allowing the algorithm to fill for a missing record which can fit the value of an explicit projection, using a table of record instances — in effect, the COQ equivalent of the terms "seen in class". Since the record inserted in the unification problem using this heuristic might itself contain missing records, this process can fire multiple times.

```
Fixpoint member (R:eqType) (x : carr
     R) (l : list (carr R)) :=
  match l with
    |h::t => if (eq R) x h then true
     else (member R x t)
    |null => false
  end.
```

Figure 1.16: List membership on an equality type

The `Canonical Structure` inference mechanism willfully constrains the generality of the unification procedure answer for ?R above. That answer is only the first registered answer out of a potential infinity of valid substitutions by some instance of a record type. Granted, a vanilla higher-order unification procedure is not able to invert record projections as is. Moreover, unifiers returned by such higher-order procedures are not always the most general anyway, as explained in § 1.2.2 on p. 38. But it is worthwhile to notice that if the unification variable ?R is constrained *before* the occurrence of the record projection equation, the system will go with that first constraint, while if it is constrained *later*, then the constraint will have to match the result of `Canonical Structure` inference for unification to succeed.

In practice, going back to our example of Fig. 1.13 on p. 33, this can be useful if we have a function defined generically on the `eqType` record type. Let us look, for example, at a member function on lists of elements equipped with an `eqType` (Fig. 1.16). When trying to apply it to a "real", concrete list of type `list bool`, we provide the following term to the system `member _ true (l:list bool)`. Given the type of `member` as present in the context, the typing rule for the product, applied to the second argument `true` generates the following unification problem:

$$\texttt{carr}\,?R \doteq \texttt{bool}$$

The `Canonical Structure` implementation is exactly a mechanism that allows COQ to solve for ?R by looking up in a table keyed on the projection name `carr`, and the projection value `bool`. In this table, if the user has entered the invocation:

```
Canonical Structure bool_eqType.
```

as the *first* declaration corresponding to the appropriate value (`bool`) for the appropriate record type (`eqType`), then COQ will substitute `bool_eqType` for ?R above. The `Canonical Structure` hint can collapse with the `Definition` of `bool_eqType` itself, but as for coercions, this is just a compact notation (Coq 2010, §2.7.15). **There also exists a `Structure` keyword to replace `Record` when defining a record that we intend to use in conjunction with this heuristic, but this is again but a notational facility**.

1.2.6 Modelling simple concepts into complex instances

In the previous process, when the `Canonical Projections` table contains a product type, COQ inserts the corresponding application with new unification variables. For example, let us define a product for the `eqType` record (Fig. 1.17).

```
Canonical Structure prod_eqType
  (a b:eqType) :=
  EqType (carr a * carr b)
  (fun x y =>
    let (a1,b1):= x in
    let (a2,b2):= y in
      eq a a1 a2 && eq b b1 b2).
```

Figure 1.17: A product instance

If we now apply `member`, as defined above (Fig. 1.16 on the previous page), to a list of pairs of booleans, the unification equation becomes:

$$\texttt{carr}\,?R \triangleq \texttt{bool * bool}$$

With notations expanded, this transforms to:

$$\texttt{carr}\,?R \triangleq \texttt{prod bool bool}$$

In reality, COQ considers that since the only value in the projection table that has a head constant matching[48] that of the right hand side of the equation, it attempts to use that instance as a solution. This syntactic matching with a head constant is a quirk we will come back to in § 1.3.4 on p. 56. The unification problem then becomes:

[48] Modulo weak head $\beta\iota\zeta$-reduction of the candidate projection value, to be precise.

$$\texttt{carr (prod_eqType}\,?A\ ?B\,) \triangleq \texttt{prod bool bool}$$

Once the left member ι-reduces we get:

$$\texttt{prod}(\texttt{carr}\ ?A\,)(\texttt{carr}\ ?B\,) \triangleq \texttt{prod bool bool}$$

This amounts to two instances of the same unification scheme:

$$\begin{cases} \texttt{carr}\,?A \triangleq \texttt{bool} \\ \texttt{carr}\,?B \triangleq \texttt{bool} \end{cases}$$

Naturally, those resulting equations are then handed back to the unification procedure, triggering new `Canonical Structure` searches. Had one of the `bool` types in the right hand side of the equation been a product instead (meaning that we would have tested membership in a list of boolean triplets) the process could have gone one step further. Of particular note is the absolutely necessary role of the initial record projection in the original term.

The takeaway moral of a close examination of the unification code remains therefore that, **COQ attempts to use computation ($\beta\iota$-reduction), first-order unification, `Canonical Structure` hints, and constant expansion (δ reduction), in that order, and recursively, to resolve unification problems**.

1.2.7 Coercions

Implicit coercions are the way COQ implements subtyping with side effects: `(x:A)` can "pass off" as a term of type `B` if there is a *coercion*, an unambiguous function `(f:A → B)`, that COQ has at hand to produce the required `(f x):B`. More precisely, the type inference algorithm can recover from a *failure* to match an argument of a computed type `A` to a required type `B` by searching for a suitable pre-declared *coercion* function of type `(A → B)`.

The way COQ deals with coercions (Coq 2010, §2.8) is slightly different from most implementations of that subtyping mechanism in proof assistants. The COQ implementation comes from the work of Saibi (1997) after he helped Bailey (1998b) achieve his own in LEGO. The differences between the final implementations of Bailey (1998a) and Saibi (1999, §5) are negligible up to idiosyncrasies of their provers. We peg mentions of work resulting from their collaboration to the name of the latter, for brevity only.

"My motivations were relatively immediate and short-term, and I consider coercions as a means to an end: a literate formalisation. A more theoretically motivated approach would probably take a more pure but less pragmatic stance towards some of the decisions that working with coercions entails. However, I started this project because I wished to undertake the literate and large-scale formalisation of some algebra over the course of my three year degree. I thus wanted to produce a working implementation that was feasible to use in practice and that was as well-suited as possible to the demands of such a project. [...] The implementation of coercions in the COQ proof-checker by Amokrane Saibi shared many of the same desires for practical checking and expressivity, and I note he made many similar choices."
(Bailey 1998a)

Implicit Syntax Saibi builds upon work realized during a project of Peter Aczel and Gilles Barthe aimed at formalizing enough Galois' theory to prove the unsolvability of the quintic (Aczel 1993). Its original question of how to engineer an overloading mechanism using inheritance[49] — rather than centering on subtyping *ex nihilo* —, finds an echo in Saibi's idea of using coercions to define an *implicit syntax* which can be automatically refined to explicit terms of the calculus. This has the benefit over in-calculus coercions to make the mechanism an obviously conservative extension, and thus theoretically lightweight. Other *implementations* of coercive subtyping have followed suit on that regard (Bailey 1998a; Callaghan 2000; Sacerdoti Coen and Tassi 2008).

[49] P. Aczel. Simple overloading for type theories. In *Types for proofs and programs*, 1994a. URL http://www.cs.man.ac.uk/~petera/overloading-for-type-theories-1994.pdf

The gist of the approach is that the *syntactic* input of a new *coercion* definition:

$$f\,[x:\mathrm{T}] = e \tag{1.3}$$

can be interpreted *incrementally* as *several* function definitions, defined for every chain of preexisting *coercions* **c**:

$$\mathbf{f}[\mathbf{x}:\mathbf{C}] = \mathbf{e}$$

Where:
- the definition of f_1,C_1,e_1 comes from the original expression (1.3) (i.e. $f_1 := f$, $\mathrm{C}_1 := \mathrm{T}$, $e_1 := e$),
- $\forall\,(2 \leq i \leq n)$, $\exists\, c_i : \mathrm{C}_i \to \mathrm{C}_{i-1}$ s.t. $f\,[x:\mathrm{C}_i] = (e_i := e_{i-1}[x \mapsto c_i(x)])$

Naturally, in the degenerate case where no preexisting coercion chain maps to T, the meaning of (1.3) is self-contained. Note the holistic view implied by inspecting preexisting coercions at method declaration (see note 51).

Coercion classes Aczel and Barthe also define a notion of class, named at first glance in reference to object-oriented practice,[50] but which aims, in their examples, at modelling algebraic structures. The "class" designation actually comes from the remark that coercions are frequently "forgetful functors", sending a child element to its parent type simply by *erasing* operations or proofs, i.e. parts of the data representation (Aczel 1994b). The

[50] *"In order to conveniently formalise mathematics in a type theory it would be useful to have incorporated into the type theory a good theory of classes in something like the sense of class that there is in object oriented programming."*
(Aczel and Barthe 1993)

term *class* is thus used for the source and target types of those coercions. Despite the vocabulary borrowed to object-oriented programming, it is the thinking that coercions function *by erasure* that is the keystone of the design.[51]

This inherently directed perspective also leads to view the whole current set of coercions as a set of trees whose nodes are classes, whose edges are coercions, and whose roots represent the local minimum "object" reached by the composed application of all "forgetful" coercions.

[51] The design of Saibi has kept the vocabulary, but, as we will see below, not even the erasure semantics. The reference to "object-oriented" vocabulary in the following paragraphs is thus a spurious artefact of history, and its meaning should be seen with a "fresh" eye.

A coherent coercion graph Saibi (1997, 1999) favors a graph, opening the possibility for bidirectional coercions. At its most timid this can mean that in the above definition, "classes" is merely the name of *those types which can be the source or target of a coercion*.[52] Moreover, the possible ambiguity introduced by *confluent paths* in the coercion graph raises interest in an invariant brought up by Barthe (1996) when extending the work done with Aczel to support multiple inheritance: the *coherence condition*.

[52] We will later give some thought on how types should group under a coercion *class*: suffice to say that by that appellation we henceforth mean precisely a node of the coercion graph.

Definition 1. *The coherence condition states that a set of coercions* Δ *is* coherent *if it contains only convertible coercions between pairs of types equal modulo convertibility, including identity at all types. That is, if for any two coercions* $c : \mathrm{A} \to \mathrm{B}$, $c' : \mathrm{A}' \to \mathrm{B}'$ *between convertible types* ($\mathrm{A} =_\beta \mathrm{A}'$, $\mathrm{B} =_\beta \mathrm{B}'$) *and any* x *a fresh variable, we have* $c(x) =_\beta c'(x)$, *and for any* $d : \mathrm{D} \to \mathrm{D}$, $d(x) =_\beta x$.

This condition makes the behavior of coercion insertion more deterministic. Indeed, the risk with an incoherent coercion graph is that the user may witness an unexpected coercion applied to his term, depending on a subtle and stateful heuristic chosing between concurrent and distinct possible coercions. All further trends of work on the subject (Barthe 1996; Callaghan and Luo 2000; Chen 2003; Luo 1997; Tassi 2008) share at least some of that concern, with variations on the exact relation used to compare two coercions.

To summarize, **at the core of implementations of implicit coercions is therefore a *coercion graph* used to somehow determine if two types are coercible during type inference, and *a concern for coherence* when adding new coercions to the graph**.

Coercions in COQ The work of Saibi extended previous solutions by adapting the notion of classes to parametrized (polymorphic) types. It also made a number of less common design decisions which give the implementation its quirks (syntactic notion of coercion classes, rigid parametric classes, strict verification of coercion graph coherence).[53]

Those choices are framed by two key decisions present in all implementations of coercions using implicit syntax.[54]

i When type inference fails to match the computed type of an argument with the declared type the receiving function accepts, the coercion mechanism should say quickly if the first is coercible to the second. Using a coercion graph that means precisely matching that argument to a node of the graph, which raises the issue of the *precision of coercion classes*.

ii Next, the issue becomes answering with a coercion function, in a

[53] Saibi also adds two distinguished coercion classes, `SORTCLASS` and `FUNCLASS`, which group under a same identifier the sorts and the members - respectively - of the function space of the calculus. We merely refer the reader to Saibi (1999, §5.4.1) and Coq (2010, §17.3).

[54] R. Pollack. Implicit syntax. In *Informal Proceedings of First Workshop on Logical Frameworks, Antibes*, pages 421–434, Sept. 1990. URL http://www.lfcs.inf.ed.ac.uk/research/types-bra/proc/proc90.ps.gz

deterministic fashion. For all implementations of coercion systems to wit, that happens using an incremental construction of a partial transitive closure of the coercion graph, accountable to some notion of coherence along the way. This raises the issue of how to *compare a coercion to preexisting ones*.

Those two design choices are each weighed against three concurrent interests:

1. The coercion mechanism aims to give a *computationally fast* answer, since it fires during the already costly mechanism of type inference (§ 1.2.2). Moreover, coercion insertion may occur repeatedly, or for different subterms during the type inference process.
2. Coercions are functions inserted implicitly, *i.e.* general functions. Since there is no particular restriction making sure they only erase a part of the data representation of their argument —the original, "intended" use — they can introduce *meaningful* side effects, something programmers have found useful on occasion. Since all this conspires to perform computation behind the back of the client user, it is important that the effect of coercions be *deterministic* - in essence, those are the stakes that the coherence condition aims at answering to.
3. Finally, the user will probably want *expressive* ways to describe and interact with the coercion graph. Specifically, we examine in the ability to specify multiple coercion paths with only slightly different domains, or to replace previous coercions by extensionally equal, but computationally more efficient versions.

The summary of those choices, as made by implementations of coercions in theorem provers, is in Fig. 1.18. Some details deserve notice: if one wants to gain computational performance with coercions, it seems more advantageous to speed up the determination of which coercion class a term belongs to, rather than to skimp on the coherence check of the coercion graph. That coercion check happens at declaration time, for which responsiveness is less of a requirement than during type inference, and practice shows that coercion graphs grow too slow in current usage to make its speed a significant issue. In fact, MATITA, for example, takes the time to do a coherence check modulo convertibility before issuing a warning when a new coercion creates conflicting paths.

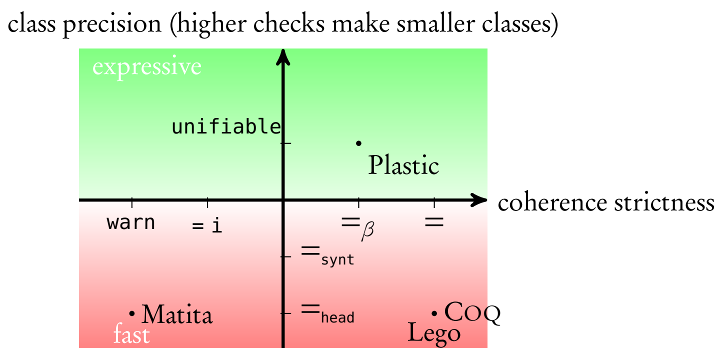


Figure 1.18: Design choices of implicit coercion systems. Determinism of behavior is not represented on that figure for technical reasons: while determinism it augments with the strictness of coherence checking, for example, some unrelated engineering choices can change that characteristic for the best in numerous cases. For example, MATITA has a `Prefer Coercion` command that can make coercion insertion more predictable using local user-specified preferences (HEL 2010).

It is remarkable that all coercion comparisons occur in the same context

of the incremental computation of a partial transitive closure of the coercion graph. Indeed, the exact verification of coherence has been shown intractable (and undecidable) in the presence of parametrized coercions (Luo 1999). Hence, when seeing a new coercion declaration c, and provided its insertion by itself does not break coherence of the graph, all systems insert the following new graph paths to a coercion table Δ:

$$\{c_i \circ c \circ c_j | c_i, c_j \in \Delta\} \cup \{c_i \circ c | c_i \in \Delta\} \cup \{c_j \circ c | c_j \in \Delta\}$$

Saibi chose to stick to a syntactic notion of coercion class mainly for performance reasons: it makes the assignation of a term to a coercion class particularly easy and fast.[55] Hence COQ coercions are keyed to some $(\mathrm{T} : \Pi\, \mathbf{a} : \mathbf{A}\ .\ s)$, with s a sort, and *the name* T — thus representing e.g. an inductive type or a constant — *is part of the key*. Moreover, the update of the coercion graph enforces a strict view of coherence: a coercion can only be added to the graph if and only if there exists no other between the two syntactic nodes it links.

[55] In common practice, this strict discipline of syntactic matching is bearable, even though the availability of unification procedures allows imagining more flexible solutions: important notions to or from which an user wants to coerce are usually named.

Uniform Inheritance condition As we have seen in the last paragraphs, whereas it is in general hard to move up the graph in the direction of more precise coercion classes, it seems comparatively easy to gain expressivity by moving from right to left. The tradeoff is avoidable for determinism, however: while relaxed coercion checks make the system less predictable, this can be mitigated by a rich user interface.

Another complication steps in when extending the notion of coercion classes to parametric types. As hinted at in the last paragraph, the approach taken by COQ is to group parametric types mentioned above by the name of their head constant. This reflects the "de-parametrized" Definition 1 on p. 46, which tests for convertibility after replacing all the parameters of the head abstraction of the coercion function by fresh variables.

As spotted by Saibi (1999, §5.5), with this coherence condition, depending on convertibility *and* syntactic parametric classes, the approach is not consistent. Notably, $p : (\Pi\, \mathbf{x} : \mathbf{A}\ .\ \mathrm{C}\ x) \rightarrow (\mathrm{D}\ \mathtt{true})$ and $q : (\Pi\, \mathbf{x} : \mathbf{A}\ .\ \mathrm{C}\ x) \rightarrow (\mathrm{D}\ \mathtt{false})$ see their ranges merged so that they are checked for convertibility, while those coercions could in fact coexist unambiguously in the same graph. In sum, **he notices that term information provided at coercion insertion should be dealt with comprehensively, or none at all: either one checks convertibility, but taking all parameters into account, or we drop parameters with convertibility checking**.

Saibi chooses to solve this by keeping the "de-parametrized" definition, but making the coherence check verify only *whether there exists a previous path (convertible or not) in the coercion graph*. An unfortunate consequence is that this does not allow defining convertible replacements for long chains of coercions: sometimes the user may want to declare explicitly that a β-reduct of a composition of coercions $c_1 \circ c_2 \circ \ldots \circ c_n$ is a coercion to reduce the size of the inferred term.[56] This also means that defining different coercions for certain parametric instances of inductive types — for example to obtain a particular behavior for instances of $\mathrm{C}\mathtt{true}$ rather than the one defined for the general $\mathrm{C}x$ — becomes harder.

[56] We will see in § 1.3.4 on p. 56 practical usage patterns where those long chains appear.

To enforce the notion of class defined over such a syntactic coercion class Saibi develops the *uniform inheritance condition*, which means that the user should replace all parametric arguments of the domain type of a parametric coercion declaration by fresh variables if he wants it taken into account by the system. In formal terms, this means that a coercion should have the type:

$$\Pi \mathbf{x}{:}\mathbf{A} \,.\, (\Pi y{:}\mathrm{C}x \,.\, \mathrm{N})$$

To mitigate the second of the two limitations we have mentioned above, he nonetheless lets the user define "identity" coercions that allow the user to define a syntactic alias for a specific parametric instance $\mathrm{C}' := \lambda(x_1 : \mathrm{A}_1, \ldots x_k : \mathrm{A}_k) \,.\, (\mathrm{C}a_1 \ldots a_n)$ with $k \leq n$ along with its translation to its parent class (the identity coercion itself). The user wanting to use the coercibility of certain terms from C' to D has to write them as members of C' explicitly — *i.e.*, the user is left the task of syntactically disambiguating the coercion classes.

Analysis and conclusions The coercion mechanism of COQ therefore has taken the syntactic approach to its fullest extent, with a view to the simplicity and efficiency of the whole mechanism. Since it occurs after a failed execution of a higher-order unification algorithm triggered by type inference, coercion insertion happens at a point where responsiveness is paramount.

However it is unclear that responsiveness requires a syntactic implementation of class assignation, or a syntactic coherence check. In practice, coercion graphs are small, and relaxing the checks of Saibi has experimentally proved tractable: without even mentioning the class assignation modulo unification of PLASTIC, the implementation of MATITA has shown that simply removing checks from that of COQ still leaves the user with an usable system.

In MATITA, coercions are equally indexed on the head of the syntactic type, but there is no uniform inheritance condition and coherence checks are indicative. Concretely that means that the myriad (in fact, the potential infinity) of distinct parameter instanciations of a given couple of $(\mathrm{C}\,\mathbf{a}, \mathrm{D}\,\mathbf{b})$ source and target parametric types coexist on the same segment between two nodes of the coercion graph.

It also means that once the coercion insertion algorithm has found a suitable class it needs to use unification to check that the coercion candidate can indeed be passed as an argument to a prospective coercion function. Saibi explicitly mentions this as an alternative to the uniform inheritance condition, but indicates being loathe to do so for reasons of incompleteness. With the hindsight the COQ community has — encompassing more that a dozen years — on the behavior of the "modern era" implementation of unification in COQ (Saibi 1999, p.94), we doubt whether this should be a serious concern any more.

Indeed, a system with such costly unification checks at coercion insertion remains usable in practice. Moreover, with operations such as user-specified coercion preference, MATITA gives greater control to the user as to the way in which candidate coercions will be attempted. As a conclusion, **changing**

Coq to close the gap with the implementation of coercions found in its cousin seems to us a desirable and yet feasible improvement.

1.3 Canonical Structures in proofs

WE HAVE DESCRIBED HOW CANONICAL STRUCTURES WORK AS A SPECIFIC COQ FEATURE, but we now want to give a hint of how they represent a *type class mechanism*, as an integrated language construct of Gallina. Thus we isolate and describe the impact of the two essential features of such a mechanism as underlined in the beginning of this chapter: constraint propagation, and automatic instances. We give examples of their use in COQ, and explain how coercions are a feature that helps integrate them by allowing smooth — though sometimes confusing — definitions for generic functions.

1.3.1 The other, older type class implementation of Coq

"Type classes are essentially implicitly passed dictionaries, and dictionaries are essentially objects. [...] Type classes are nice. A cottage industry of Haskell programmers has sprung up around them."

(Martin Odersky, talk to IFIP WGP '06, Boston)

As hinted at since the beginning of this chapter, the *type classes* of Haskell (Wadler and Blott 1989), the *traits with implicits* of Scala (Oliveira et al. 2010), the *concepts* of C++ (Gregor et al. 2006) and BitC (Shapiro et al. 2008), the *generalized interfaces* for Java (Wehr et al. 2007), and a flurry of other recent incarnations in theorem provers (Asperti et al. 2009; Haftmann and Wenzel 2007; Sozeau and Oury 2008), are a way to *group* values under an elaborate type, and write procedures on this index. We call them type class mechanisms, or *concepts*.

With COQ, doing this sometimes looks slightly magical because it permits to seemingly generate values "out of thin air". We can write, say, a generic function that takes a list and an index, and returns the following:
- the value of the element at the position given by the index
- or if the index is out of the lists' bounds, another value of that same type.
All this happens in a type-safe manner but without apparent type-level exception handling. This sort of thing tends to make the axiom-detection-sense of the reader tingle:

```
Definition nth_nat (n : nat) (l : list nat) := nth n l.
Definition nth_bool (n : nat) (l : list bool) := nth n l.
```

`nth` is polymorphic, but no axiom declarations are involved, of course. The trick consists, when defining `nth`, in swapping the type variable we would use if we were defining a polymorphic function - here, the type of list elements - with a record type that provides the sorely needed default value for computing `nth`.

When calling `nth`, the user only supplies part of the record. A mechanism for recalling which values of that record type the compiler is aware of then enters into play, and picks the right one. There are *two* such mechanisms in COQ: if the user chooses `Class`es, those pre-registered values are `Instance`s, and the definition of `nth` looks like Fig.1.19(a). If he chooses `Structure`s, they are `Canonical Structure`s, and `nth` is as in Fig. 1.19(b).

In both occurrences, a flavor of *inheritance* complements this *instanciation* mechanism, so as to extend such record types by reference: Sozeau (2008, §7.2) covers it for `Class`es, and Garillot et al. (2009) details best practices for `Structure`s (we will come back to this subject in 1.4). This second feature, inheritance, achieves making `Class`es and `Structure`s proper *first-*

```
Class defaultType (T : Type) :=
  default : T.
Instance nat_defaultType
  : defaultType nat := 0.
Instance bool_defaultType
  : defaultType bool:= false.

Section Nth.
Variable T : Type.
Fixpoint nth '{dT : defaultType T}
  (n : nat)
  (l : list T) : T :=
  match l with
    h::t => match n with
             S n0 => nth n0 t
             | _ => h
           end
    |_ => default
  end.
End Nth.
```

(a)

```
Record defaultType :=
  { sort : Type; default : sort }.
Canonical Structure natdefaultType :=
  Build_defaultType 0.
Canonical Structure booldefaultType :=
  Build_defaultType false.

Section Nth.
Variable dT : defaultType.
Fixpoint nth (n : nat)
  (l : list (sort dT)) : (sort dT) :=
  match l with
    h::t => match n with
             S n0 => nth n0 t
             | _ => h
           end
    |_ => default dT
  end.
End Nth.
```

(b)

Figure 1.19: Pure Coq generic index selection: (a) describes the syntax for `Classes`, (b) for `Structures`.

class type class implementations. We will now put `Class`es aside (until a brief comparison in § 1.4.5 on p. 79), and focus on `Canonical Structure`s.

When `nth` is called (Fig. 1.20), it receives a *returned type*, `(list int)` for the list argument. During type inference (or at the application of a tactic), this type is reconciled with the *expected type* specified in he definition of `nth`, using unification. Since this expected type involves a *record projection* (`sort`), the compiler finds a solution for `?x` in the following unification equation:

```
sort (?x:defaultType) = int
```

Notice that this problem has no solution using a vanilla higher-order unification algorithm. The `Structure` mechanism consists in updating a table of special solutions, keyed by projection's names and return types, at each `Canonical Structure` declaration the user makes, and in making those solutions available to the unification procedure. Once a fitting record is found for `dT`, the value for `default dT` is obtained by deferring to the general unification procedure.

```
Eval compute in
  nth 2 (1::1::42::1::nil).
>      = 42
Eval compute in
  nth 5 (1::1::42::1::nil).
>     = 0
```

Figure 1.20: Application of either 1.19(a) or 1.19(b).

1.3.2 *Constraint propagation*

For a concept mechanism, constraint propagation means that when using an algorithm that takes as argument a concept that is defined *by refinement* of another, the algorithm should have access to the members and operations specified by that other. Naturally, this reference to refined parents needs to carry over to *instantiation*, where the user should not need to pass explicitly a complete chain of objects — where the interpretation of each as concept instance would be *a posteriori* seen as a refinement of the previous one — simply to effectively use a generic function.

An object-like system If this last paragraph sounds like inheritance support in an object-oriented language, it is not a coincidence: the similarities are numerous, inherent and well-known (Cook 2009; Kiselyov and Lämmel 2005). In a nutshell, a concept mechanism offers a measure of *procedural abstraction*, since its goal is to specify virtual operations, distinguished el-

ements — and occasionally, proofs — defined by reference to — at most — a few common, *abstract variables* — and most often, types. In Haskell, for example, this last element is the single *type parameter* of the class declaration. In *multi-parameter* implementations of type class systems (such as the eponymous Haskell extension, MATITA, or COQ), those abstract type references can be many. In dependent type systems (COQ), they can sometimes be values. Nonetheless they remain *abstract* variables: synchronization points on which to implement a multisorted algebra of operations sharing those references. This does hint to a similarity with *object interfaces*: concepts allow an algorithm to operate on any value that has the necessary operations.

This is not to say concepts are by any means equivalent to objects — even up to implementation-specific idiosyncrasies. We refer the reader interested in the comparison to the aforementioned references, but to get a sense of a striking difference, it probably suffices to mention that type classes often lack any kind of mechanism to prevent the programmer from accessing the representation of a given instance (Cook 2009, §5.3).[57] But it remains along the lines of that similarity to expect concepts to have an instantiation that takes refinement into account, and only require the user to provide a model of the most refined concept for each inheritance chain to compute or reason with a generic algorithm. In practice, for `Canonical Structure`s, this seems to mean — at first glance — specifying concepts using records in *telescopic* style (§ 1.1.10 on p. 32).

[57] Apart from type class systems developed on top of an object-oriented type language (e.g. Scala), where the programmer can leverage the containment facilities offered by the object system to achieve representation independence. Advanced instantiation mechanisms such as well-scoped overlapping instances (Scala), or the declarative model of `Class`es based on the loose `eauto` tactic, can also mitigate the autognosis problem. We will come back briefly to overlapping instances in § 1.4.5 on p. 79.

Refinement and projections Notwithstanding the *value classes* mentioned above, let us look at the more frequent case of concepts indexed by an abstract type, for the sake of discerning usage patterns. There are at least two natural points of entry to a `Canonical Structure` inference chain.[58] The first, that we discussed extensively since § 1.2 on p. 37, is by applying a generic function or lemma. The inference proceeds in this case exactly as in the example of `member`, treated extensively in § 1.2.5 on p. 43: the `Canonical Structure` is computed on the first projection of the record, that which returns the carrier type. Sometimes, however, the user favors one of the *other* projections to trigger inference. This use case works particularly well with notations, and is just a more general case of the above: generic functions usually require an instance of a record only because they make use of one of the operations they contain.

[58] What we mean by *natural* will be more understandable after the use of *phantoms* we will expose in § 2.3.2 on p. 108, and even more afterwards. The word "common" makes an acceptable alternative in the meanwhile.

Let us consider again the lattice of Fig. 1.12 on p. 32 and 1.13 on p. 33. We can define a notation that will unfold to the direct use of one of the computational members of the record as in Fig. 1.21. In telescopic style, we see that the presence of a projection as applied to the parent record is guaranteed.

```
Notation "x == y" :=
  (eq_op x y)
  (at level 70 ): bool_scope.
Lemma eqP : ∀T, ∀x y : T,
  reflect (x = y) (x == y).
  Proof.
  by rewrite /eq_op; case=> ? [].
  Qed.
Lemma eq_refl :
  ∀(T : eqType) (x : T), x == x.
  Proof.
  by move=> T x; apply/eqP.
  Qed.
```

Figure 1.21: Canonical structure inference through the use of an arbitrary projection operator.

In fact, the existence of those two entry points has consequences for the rest of inference. Let us suppose we want to define a *child* concept, that refines its *parent*: in a typical case, the parent would specify a carrier type, a constant (such as an algebraic operation) and a property, that the child would supplement by adding, say, an additional operation and an additional property.

- A generic function defined on such a record instance accesses the effec-

tive computational content (namely, the element having the carrier type of the oldest ancestor) by a chain composed of the successive first projections (customarily assigned to the carrier) of each record. In our case, that generic function f looks like:

```
Definition f (r: childType) (x : p_carrier (c_carrier r)) := ...
```

- The operations defined in the child are likewise defined by reference to computational content extracted from first element of the record, as in Fig. 1.13 on p. 33. The member to which the refined operations lead likewise contains a projection chain similar to the above.

This ensures that in the process of unification, the record instance inferred from the parent, once (eagerly) ι-reduced, always exposes the next projections necessary to trigger the inference of the child(ren): COQ's unification proceeds depth-first.

For Canonical Structures, one cannot hope to get constraint propagation through parametric reference. **Record parameters can be present in instances, and retrieved by unification, but themselves they contain no projection to allow the Canonical Structure inference mechanism to support inheritance**.

Eventually, the fact that we have to resort to telescopes can at first seem worrying, for reasons mentioned in § 1.1.12 on p. 34. Fortunately, it turns out that the inefficiencies of the telescope pattern can be addressed, through a rather involved pattern called *packed classes*, which moreover allows us to model more complex relations, such as multiple inheritance. We will explain later (§ 1.4.1 on p. 58) how we used it to overcame the problems we described.

1.3.3 *Automatic Instances*

I would now like to dispel a number of shortcuts made in explaining type classes which undercut the crucial importance of what is permitted by automatic instance generation in general, and Canonical Structure inference through unification in particular.

Stricto sensu, type classes are not just about the definition of type-indexed generic operators, despite being sometimes presented as such. That notion is older than even the earliest flavor of type classes (Kapur et al. 1981), and there are numerous ways to achieve this objective without reference to concepts, as the bounded polymorphism of generic flavors of Java, C++ templates, or polymorphic records have amply shown.

Another approach remarks that type classes indeed provide one solution to the famous *expression problem*.[59] The expression problem is a benchmark of expressiveness and modularity for programming languages, that consists in extending a data type (in the classical example, a type for arithmetic expressions) with both a new case (multiplication, for example), and a new operation (such as pretty-printing the expression). The extension should be done without changing or recompiling the original code, and preserve static type safety. It was found a challenge for both object-oriented and functional paradigms.

[59] (Wadler 1998), also called *extensibility problem* by Findler and Flatt (1998).

But if the requirement for being a solution is the minimum of *somehow* allowing for extensions in both typed data and methods, however crufty,

then, so are the visitor pattern, modules, and polymorphic variants, in increasing levels of refinement.[60] We believe with Lämmel and Ostermann (2006, §2.4) (and Swierstra (2008)), that type classes (and respectively type classes with overlapping instances) can solve that problem well, but other improvements suggested in our own dependently-typed context (§ 1.4.4 on p. 74), make us contend that this angle does not capture a type class mechanism in all generality.

[60] See (Odersky and Zenger 2005) for a survey of some of the older solutions to this problem.

Another common mention of type classes presents them in a manner similar to how Phil Wadler does in the following:

> In general, saying that a type variable extends an interface (which usually is parameterized over the same type variable) in Java serves the same role as saying that the type variable belongs to a type class in Haskell.
>
> (Biancuzzi and Warden 2009)

But while this is close enough as a transitory analogy for of a type class mechanism, it does not describe it exhaustively. As Chakravarty et al. (2005) remarked when using OCAML modules to transpose type classes, the latter also consists, inherently, in automatically passing a dictionary of values when a generic method is called. They found having to do this passing manually particularly tiresome:

> Furthermore, we showed that Haskell type classes can be translated into ML modules by using first-class structures as runtime evidence for type class constraints. [...] It is not recommended writing programs in the style of the translation by hand because too much syntactic overhead is introduced by explicit dictionary abstraction and application.

Kiselyov (2007) came to the same conclusion when he emulated type classes in OCAML using, this time, polymorphic records:

> Although the OCaml implementation is a faithful translation of Haskell code, the explicit use of dictionary passing is quite an annoyance.

One fact of note in proofs of algebra, as present in the SSReflect archive, is that some instances can be impressively large, even though the terms they are inferred from are themselves tractable. This should not be surprising: since we are dealing with finite group algebra, it is frequent happenstance to encounter *algebraic expressions*. A midsize instance would be the third isomorphism theorem:

```
Theorem third_isom :
  {f : {morphism (G / H) / (K / H) ↦ coset_of K}
    | 'injm f
  & ∀A : {set gT},
    A ⊆ G → f @* (A / H / (K / H)) = A / K}.
```

The left member of the final equation is a group. This is ensured by three applications of a construction (/) that is *proven to send* the group structure of two groups to another group structure,[61] and one application of a function (@*) *proven to send* a group and a group morphism to the group formed by the image of the former by the latter. Asking the user to remind this regular decomposition to the prover, composing those proofs by hand, each time he wants to apply a lemma depending on the group structure of that left-hand side, is plainly untractable.

[61] Please ignore the normality of divisor groups, here. The particular way the library deals with this issue is explained in (Gonthier et al. 2007, §3.3).

The answer of a true *type class implementation* to this problem is a transparent representation of the principle of compositionality, namely the principle that the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them. Using type class mechanisms, and in particular `Canonical Structure`s, the user is provided with an engine which takes meta-information (i.e. record instances) registered on an algebra of atomic objects as well as on compositions rules for those kinds of objects, and propagates this information *automatically* to any appropriate algebraic expression obtained from those atoms and composition laws (see the example in 1.2.6 on p. 44).

1.3.4 Coercions and Structures

By themselves, coercions are a well-know feature that has no obvious inherent relation with a type class implementation. And indeed, the two mechanisms can be understood and used in perfect isolation. However, Saibi implemented them in COQ conjointly with `Canonical Structures` (Saibi 1999, §5), leading to a tight integration which favors specific usage patterns. For instance, the `:>` symbol inserted in a `Record` member declaration declares the corresponding projection as a coercion. Since the presence of a record projection in the arguments of a function is necessary to trigger `Structure` inference, this greatly helps the legibility of generic lemmas and functions which use `Structure`s. We give an example in Fig. 1.22[62].

```
Record defaultType :=
  { sort :> Type;
    default : sort }.

Section Nth.
Variable dT:defaultType.
Fixpoint nth (n : nat)
  (l : list  dT ) :  dT :=
  if l is (h::t) then
    if n is S n0 then
      nth n0 t
    else h
  else default _.
End Nth.
```

Figure 1.22: `nth` in SSReflect, using coercions. (compare to 1.19(b))

[62] Until the *packed classes* design is described in § 1.4.1 on p. 58, we will highlight all points where coercions are implicitly present in code snippets.

One should note that as mentioned above, the assignation of a term to a coercion class is purely syntactic, hence COQ registers *all* record projections in a `Canonical Projections` table provided (i) the record member is *named*, (ii) the projection *value* (`nat` in our example) has a *head constant*, and (iii) that same *head constant* was not previously registered as a value for the *same* projection.

An even more important point of this usage is that it furthers blurs the distinction between functions operating on a record and functions operating on the projections of that record.

Let us consider the definition of a vector-like type, as done in the SSReflect library, represented in Fig. 1.23 on the next page.[63] `tuple_of` is the declaration of a finite list whose type exposes its size as a parameter. The instances `nil_tuple` and `cons_tuple` build, given a list, a record instance exposing the suitable parameter. The use of `Canonical Structure` instance search to do this is the equivalent — greatly simplified by the availability of dependent types — of the classic trick of exposing the structure of data at the type level usually shown with Haskell type classes, and famously exposed by Hallgren (2001) and McBride (2003, §3.1).

[63] See also (Gonthier and Mahboubi 2010, Exercises 6.2.1-4). Note this is an example of a *value class*: a `Structure` indexed on a parameter, here the first projection `tval`, which is not of kind `Type`. The `seq` type is the SSReflect flavor for (the usual, polymorphic) lists, and `size` is the function which returns their length in the expected manner, of type `∀(T:Type), seq T → nat`.

An interesting consequence of this approach is that rewriting using a lemma such as `∀(x : n.-tuple T), size  x  = tsize x.` on the sequence `[1;2;3]` poses no problem: the `cons_tuple` record is found in the `Canonical Projections` table as a substitute for `?` by unification, in the equation `tval ? = [1;2;3]`. The application of `size` indeed hides the frequent shorthand of designing the image of a record by one of its *canonical projections* (`tval`), using the name of the record itself, as permitted by the *coercion* indication (`:>`) in `tuple_of`. The coercion insertion that triggers the use of this projection

```
Structure tuple_of (n : nat)
  (T : Type)
  : Type := Tuple {
    tval :> seq T;
    _ : size tval == n
    }.

Definition tsize (x: tuple_of T n) :=
  n.

Notation "n .-tuple" :=
    (tuple_of n) : type_scope.

Canonical Structure nil_tuple T :=
   Tuple
   (erefl _ : @size T [::] == 0).
Canonical Structure cons_tuple n T x
   (t : n.-tuple T) :=
   Tuple
   (valP t : size (x :: t) == n.+1).
```

Figure 1.23: Finite `tuple`s with type-level size

comes from type inference failing to reconcile `seq T`, the expected type of `x` as deduced from the definition of `size`, and its computed type `n.-tuple T`. The remainder of type inference, after coercion insertion, results in the aforementioned equation.

This practice of defining functions on implicitly coerced arguments is adopted with a matter-of-fact and pervasive approach when dealing with `Structure`s in the SSReflect library. It is in fact encountered often in the more common case of *type* classes than in this example. When `Structure` inference fires from the use a *value* obtained by a projection `p` whose range has kind `Type`, i.e. when the (generally first) member of the record `recordType` is a `(T:Type)`,[64] it is very frequent to see that generic functions use that projection value implicitly, while looking like they compute on the whole record instance, as in Fig. 1.22 on the facing page. In that case, a generic function definition such as

```
Definition f (r: recordType) : rangeType := ...
```

with apparent type `(recordType → rangeType)` will in fact be interpreted as a term of type `(p recordType) → rangeType`, i.e. `(T → rangeType)`. `f` looks like it operates on a record instance, but in fact, if we just give it something that can be inferred as the *first projection* of a *canonical instance*, it "works".

[64] This also applies for compositions of projections : in telescope-style, the first projection of such a `Structure` would be a parent `Structure` projecting to a `Type` (up to composition of first projections).

On the other hand, the definition of `tsize` shows an example of a function that can *not* be passed a simple sequence. Its first argument is a member of a value class, *i.e.* a record type not inherently coercible to `Type`. This particularity is a sufficient (but not necessary) condition to ensure that the type of the function we are defining will require a *full record instance* as an argument, without using inference-triggering projections. Hence, using a sequence `s` with a function operating on the output of `tsize` like above is more complex: we would like to pass to `tsize` "the `tuple_of` instance for a `(s : seq)`" implicitly — a non-obvious problem we will address in section § 2.3.2 on p. 108.

When we define a function whose argument effectively needs to be a `tuple_of` record, it is difficult to use it on concrete tuple instances, namely sequences, and deceptively so: it is well possible to realize that constructions do not operate on *projections* of the structure well after the (systematic) stage of formalizing what happens when they are passed *instances* of the structure. **Hence, in general, it is useful to remember that generic functions tailored to trigger `Canonical Structure` inference are best defined on projections of the record type.**

1.4 Ecosystem and Improvements

IN THIS SECTION, WE AIM AT PRESENTING EXTENSIONS of the fundamental model of concepts based on `Canonical Structure`s that we have explained until now. Firstly, we solve the expressivity and efficiency issues of telescopic-style `Canonical Structure`s by suggesting a new model called Packed Classes. After a thorough analysis of the benefits of this model, we demonstrate how it can express previously-thought intractable structural relations : more specifically, we treat *multiple inheritance* on the basis of a challenge by Spitters and van der Weegen (2011) and we show that we can emulate the Pebble-style definition of a (categorical) adjunction as proposed by Sozeau and Oury (2008). Then, we move on to further creative uses of the `Canonical Structure` mechanism in COQ: we explain how to direct inference along *instance chains*, a sort of a continuation-passing semantics for automatic instantiation.[65] To conclude this tour of the state-of-the-art of packed classes in COQ, we provide a comparison with the type class implementation of Haskell, and the `Class`es of (Sozeau and Oury 2008).

[65] We will suggest how to apply this method to implement ad-hoc reflection of a λ-calculus with terms of COQ in § 3.4.4 on p. 134.

1.4.1 Inheritance mechanisms : packed records

Telescopic Issues The problems with telescopic-style programming with records are in fact more numerous than the complexity problem described in § 1.1.12 on p. 34 — even though it remains the most pressing issue for dealing with concrete algebraic objects, which are often nested. Let us look at those additional inconveniences. For starters, it is unclear how to model multiple inheritance using telescopes: if both parent records are mentioned in the child, that means that an instance of the child contains two unsynchronized copies of the data it would want to refine. And since a child instance can appear only once in a given proof context, it will have to appear as *only one* of all projection chains sending it to a carrier type. This single projection chain will dictate the only parent structure that will be automatically inferrable on that object.

Secondly, when successively refining the `Canonical Structure` one can bestow upon a given hinting term, the more refined we want to be, the more we have to add projection applications to that term. As mentioned in § 1.3.4 on p. 56, this is, in the most common cases, done implicitly for the user using the coercion mechanism, but it is particularly inefficient for technical reasons.[66]

[66] Notably:
- Some functions over these projection chains are exponential over their size, notably term comparison (as mentioned in note 30 on p. 34).
- The enforcement of a single coercion for a given (source,range) couple of coercion classes (§ 1.2.7 on p. 45) means that we cannot simplify this projection chain by allowing for a shorter, distinct but convertible function to be inserted in cases where it will not be used for triggering `Structure` inference. In the typical case where one just wants to coerce to the carrier type of the instance, this is inefficient.

In fact, as we have already remarked (§ 1.3.2), `Canonical Structure` search is triggered by either by using a generic function which takes a structure instance as argument, or by using a structure-specific operation. Both entry points feature the projections necessary to request the appropriate inference from the `Canonical Projections` table. Two key insights allow to improve the behavior of the inference procedure from what the telescopic-style allows:

(i) inheritance can be de-coupled from the use of a record projection to infer a `Structure`.

(i) the result of a `Canonical Structure` inference using projections, i.e., the

non-ι-reduced application of the projection function to the record instance, forms itself a new term, on which specific `Canonical Structure` hints can be declared and used. In particular, *having detected a child structure provides a specific constant on which a* `Canonical Structure` *for the parent can be declared.*

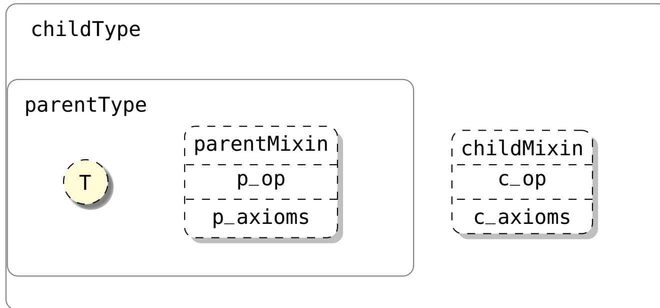


Figure 1.24: Organization of a parent-child refinement in telescopic mode

To make these intuitions clearer, let us suppose we have a parent structure `parentType`, with a first projection `p_sort` to a carrier type of sort `Type`, and a binary operation on that carrier that we will call `p_op`. Let us also suppose that this `Structure` is refined by a `childType`, adding a specific operation `c_op`. In telescopic-style, this means that the first projection `c_sort` of `childType` will be of type `(childType → parentType)`, as in Fig. 1.25 and 1.24.

Let us look at structure inference, assuming both child and parent have instances over the natural numbers. Let us call these instances, respectively, `int_childType` and `int_parenType`. We look at a first example, say $2 \triangle 2 \doteq 4$, where $\doteq$ is a relation defined by the `parentType` structure, and $\triangle$ is an operation defined by the child. The rundown of inference is on Fig. 1.26 on the next page.

```
Structure parentType : Type := {
  p_sort : Type;
  p_op :  p_sort  →
          p_sort  →
          p_sort
}
Structure childType : Type := {
  c_sort : parentType;
  c_op :  c_sort  →
          c_sort  →
          c_sort
}
```

Figure 1.25: An abstract inheritance situation in telescopic style

We can see that the order in which the `Canonical Structure` lookups are made are suboptimal. The way we set up projections as inserted by coercions forces COQ to look for structures from the outermost coercion to the innermost one. Translated to telescopic style of inheritance, this means that we are looking for an arbitrary level of structure refinement by successively trying to see an object as an instance of all the less refined structures we can bestow him, from the least refined to the most refined.

In the practical case of library development, this is ineffective for two reasons:

The size of the search domain A *lot* of objects possess the simplest structures of the hierarchy, hence the search space is larger for the projections tied to the least refined structures. It seems wasteful to make this the most common case.

The length of the projection chain When `Canonical Structure` inference looks for an instance of a type refined n times, it will fail after having possibly succeeded up to $n-1$ times. We would like to see a paradigm in which it fails after a single instance lookup.

Let us remind that the goal of inferring the type of the expression $(2\triangle 2) \doteq 4$ is to provide the right implementations for the overloaded symbols $\triangle$, $\doteq$. The first step towards type inference is to meta-expand notations and implicit arguments. The explicit term we want to type is therefore:

$$\texttt{p_op}\,?\beta\,(\texttt{c_op}\,?\alpha\,2\,2)\,4$$

Taking into account the implicit coercions inserted at the declaration of the `childType` `Structure`, the type of `c_op` is

$$\texttt{p_sort}(\texttt{c_sort}\,?\alpha) \to \texttt{p_sort}(\texttt{c_sort}\,?\alpha) \to \texttt{p_sort}(\texttt{c_sort}\,?\alpha)$$

This type provides the system with the following two equations, the second of which is flexible-flexible. Conveniently, it is because COQ proceeds with unification meta-variables in a left-to-right fashion that this second equation is solved last.

$$\texttt{p_sort}\,?\gamma \triangleq \texttt{int} \qquad (1.4)$$

$$\texttt{c_sort}\,?\alpha \triangleq\, ?\gamma \qquad (1.5)$$

The typing of `p_op`, again involving coercions, gives the two other equations:

$$\texttt{p_sort}\,?\beta \triangleq \texttt{p_sort}\,?\gamma \qquad (1.6)$$

$$\texttt{p_sort}\,?\beta \triangleq \texttt{int} \qquad (1.7)$$

Of those two new equations, (1.6) is a flexible-flexible pair (which again has no operational consequence), and (1.7) involves the second argument to `p_op`. COQ treats disagreement pairs in strict left-to-right argument order, and depth first, so the first equation `Canonical Structure` inference will deal with is:

$$\texttt{p_sort}\,?\gamma \triangleq \texttt{int}$$

COQ therefore looks up a `parentType` structure whose `p_sort` projection matches `int`, and finds `int_parentType`. Since the `Canonical Structure` table is keyed by projection, we represent this by the notation:

$$\texttt{lookup}(\texttt{p_sort}, \texttt{int}) \rightsquigarrow \texttt{int_parentType}$$

COQ then propagates this to the rest of the first argument to solve the second equation:

$$\texttt{lookup}(\texttt{c_sort}, \texttt{int_parentType}) \rightsquigarrow \texttt{int_childType}$$

Once this is performed, COQ just has to simplify the third equation we mention to equalize $?\gamma$ and $?\beta$, substituting them both with `int_parentType`, and solving the unification problem entirely.

Figure 1.26: Telescopic-style structure inference

Packed Classes To solve this problem, we propose the following design, based on a strict separation between mathematical and usability concerns, and called *packed classes*.[67]

- Each new piece of structural information — representing a set of new operations and properties we want to treat as an atom — is packaged in a *mixin*, a record parametric in the target carrier type (and possibly in other parent components, themselves parametric in that carrier type — as needed in the definition).
- For each specific *structure* we want to define, we group all necessary components (i.e. its *mixins*) in a single record, again parametric in the target carrier type, called its *class*.[68] Often this is just a reference to the parent type (the parent's *class*), plus a single mixin defining the additional properties specific to the child. Occasionally, when we define a base type which needs no parent but defines all its properties from scratch, this is a mixin by itself.
- Finally, the *structure* we work with is defined as a package, which has the carrier type as first projection, and for which its *class*, bound to that projection, is the other member of the record.

An example is given in Fig. 1.27 on the facing page and 1.28 on the next page. In the first figure, we can see a base type, `parentType`, which does not need to make reference to any other previously defined structure. Hence, we merge its *class* and its the *mixin* of its "proper" members. Inheritance comes in when `childType` is defined by reference to `parentType`, so that

[67] Not to be confused with COQ's *class* mechanism

[68] This is the construction that gives the *packed classes* its name, by reference to HASKELL's type classes.

we see a childTypeClass defined by glueing its specific *mixin* with the *class* regrouping the properties of its parent.

The roles of the three "packed classes" declarations are the following:

- The *mixin* provides an abstraction for a consistent bundle of members and properties: a compositional atom that possibly exposes some parameters it depends on.
- The *class* regroups abstractions that form an entity in an object hierarchy. In particular, it helps defining structures by refinement.
- The *type* allows the inferrability of said class on a given object using the Canonical Structure mechanism.

The design of these Structure declarations is systematic. Notably, we can declare a Canonical Structure for the parent *structure*, generically, from *any* instance of the child. Moreover, it is possible to define structure-specific aliases for the projections to each interesting operation of such a packed record, recovering an *application programming interface* (API) close to that of the telescopic style. Since we already redefine specific lemmas for all *properties* included in a Structure anyway (note 22 on p. 28), this is not much of an additional hindrance.

In the SSReflect library, it is thus customary to define such inheritance cases in Modules, formatted after a common syntax to present code which differs minimally from one case to the other.

```
Structure parentTypeMixin (T) := {
  p_op : T → T → T
}

Definition parentTypeClass (T) :=
  parentTypeMixin (T).

Structure parentType := {
  p_sort : Type;
  class : parentTypeClass(p_sort)
}

Structure childTypeMixin (T) := {
  c_op : T → T → T
}

Structure childTypeClass (T) := {
  p_class : parentTypeClass(T);
  c_mixin : childTypeMixin(T)
}

Structure childType := {
  c_sort : Type;
  class : childTypeClass(c_sort)
}

Canonical Structure child_parentType
    (c:childType) :=
  build_parentType (sort c) (p_class
    (class c)).
```

Figure 1.27: An abstract inheritance situation in packed classes style

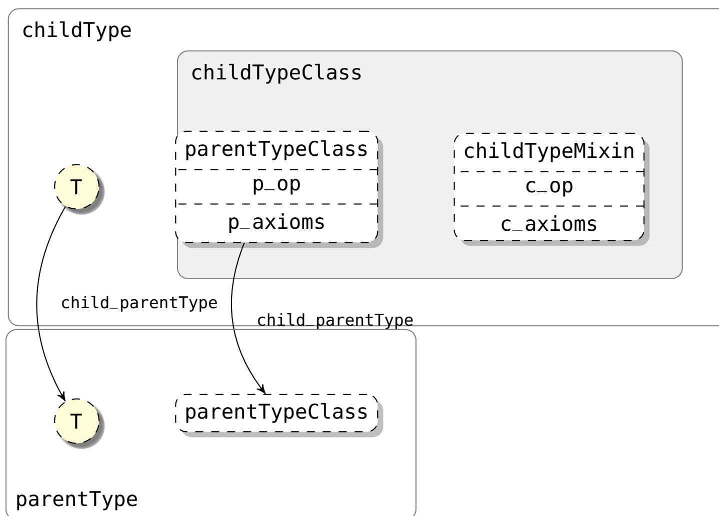


Figure 1.28: Organization of a parent-child refinement in packed classes style

We can now look at how inference is performed in the case of packed classes, taking again our example of $2 \triangle 2 \doteq 4$. The rundown of type inference is presented in Fig. 1.29 on the next page. The trick consists in reusing the form of the result of the first inference, triggered by the type of c_op, seen as the target value during the second inference, triggered this time by the type of p_op. Indeed, as we have mentioned already (§ 1.2.6 on p. 44), Canonical Structure inference does only head constant comparison for a given projection, and proceeds depth first. The relevant information of the result of the first inference is thus c_sort, *i.e.* only the indication of what *type* of structure this is a value of. But this is enough because when we look, immediately after, for a parentType structure to match a value starting with c_sort, there is an unambiguous answer to provide: the Canonical value

given by `child_parentType`.

For the determinism of inference, it is crucial to obtain an identical structure for the variable $?\beta$ if the arguments of `p_op` are reversed, that is if we are looking looked for a type for the expression $4 \doteq 2\triangle 2$. Because `child_parentType` only recombines components that form part of all parent instances (as guaranteed by our discipline), it returns a value that is *convertible* to the structure obtained by inferring a `parentType` on integers : the code of `child_parentType` is agnostic in the instance-specific contents of `childTypeMixin` or `parentTypeMixin`.

As usual, the first step in inferring the type of that expression is to unfold notations. We obtain the same typing goal, modulo renaming of the relevant projections, as mentioned above.

$$\texttt{p_op}\,?\beta\,(\texttt{c_op}\,?\alpha\,2\,2)\,4$$

Adopting as notations:

$$\texttt{p_op}\,?\beta = \texttt{p_op}(\texttt{class}\,?\beta) \quad (1.8)$$

$$\texttt{c_op}\,?\alpha = \texttt{c_op}(\texttt{c_mixin}(\texttt{class}\,?\alpha)) \quad (1.9)$$

What is most remarkable, though is the type of those aliases, each time involving a single projection, characteristic of the structure specific to the operation:

$$\texttt{p_op}:\texttt{p_sort}\,?\beta \to \texttt{p_sort}\,?\beta \to \texttt{p_sort}\,?\beta \quad (1.10)$$

$$\texttt{c_op}:\texttt{c_sort}\,?\alpha \to \texttt{c_sort}\,?\alpha \to \texttt{c_sort}\,?\alpha \quad (1.11)$$

This time, hence, the type of `c_op` enforces a rule involving a single projection:

$$\texttt{c_sort}\,?\alpha \triangleq \texttt{int}$$

With its two distinct arguments, the type of `p_op` generates the constraints:

$$\texttt{p_sort}\,?\beta \triangleq \texttt{c_sort}\,?\alpha \quad (1.12)$$

$$\texttt{p_sort}\,?\beta \triangleq \texttt{int} \quad (1.13)$$

As previously, because of the left-to-right, depth-first resolution of argument unification, `Canonical Structure` inference will proceed with the first equation:

$$\texttt{lookup}(\texttt{c_sort},\texttt{int}) \rightsquigarrow \texttt{int_childType}$$

It is the resolution of the resulting equation (1.12) that becomes most interesting. Since version 8.3, Coq does not eagerly ι-reduces the result of the first `Canonical Structure` lookup any more. This gives us the opportunity to work with an unusual redex, characteristic of the inference of a previous, more refined `Structure`. Thankfully, that allows us to reuse exactly the instance we had defined for that case during the `Structure` declaration (Fig. 1.27):

$$\texttt{lookup}(\texttt{p_sort},\texttt{c_sort int_childType}) \rightsquigarrow$$

$$\texttt{child_parentType int_childType}$$

It is only at a further reduction step that Coq simplifies the obtained `Structure` instance into `int_parentType`, obtaining exactly what we expect. We will come back on the sensitivity of the process to definitional variants in § 1.4.4 on p. 74, and § 2.3.2 on p. 108.

Figure 1.29: Packed Classes style structure inference

Advantages We provide an example of the packed classes technique: the reinterpretation of the previous `Structure` declarations leading up to our lattice in Fig. 1.31 on p. 66. The considerable amount of boilerplate goes by with users of the SSReflect library as a systematic way to ensure the usability of their definitions, as it is tailored to be easily reproducible.[69]

To ensure a fair comparison with the telescopic style, and — even more importantly — with the pebble style, we do not use implicit arguments in that example, even tough it would be customary. However, this snippet showcases other typical patterns of the SSReflect library:

- The parsimonious naming of projections, e.g. for the `class_of` member.
- The unassuming use of coercions (that we do henceforth do not highlight any more). We prefer the explicit `Coercion` declaration to the hard-to-spot :> abbreviation.

[69] For now, we kindly ask our reader to ignore the workings of the `clone` and `pack` functions. Let us simply state that `clone` is a general constructor used with notations to return a copy of a canonical instance of the structure defined in the current module, given one of its projection. `pack` is a customized constructor that works conjointly. We will address their semantics in detail in § 2.3.4 on p. 114.

- The use of the inferred instance of a child structure to deduce parent instances (marked by the word “Inheritance” in comments).
- The use of `Module`s to structure name space and to reuse (ideally, copy-paste) code patterns.
- Finally, because of the way COQ compares records, starting sometimes with the last member, and sometimes with the first member, it is an easy optimization to *repeat* the carrier type at both ends of this final, non-parametric `Structure` — though this has no deep theoretical meaning.[70]

[70] This document makes the choice of showing runnable, practical tricks used by the Mathematical Components team — see also note 33 on p. 35, for example. That includes the less elegant ones.

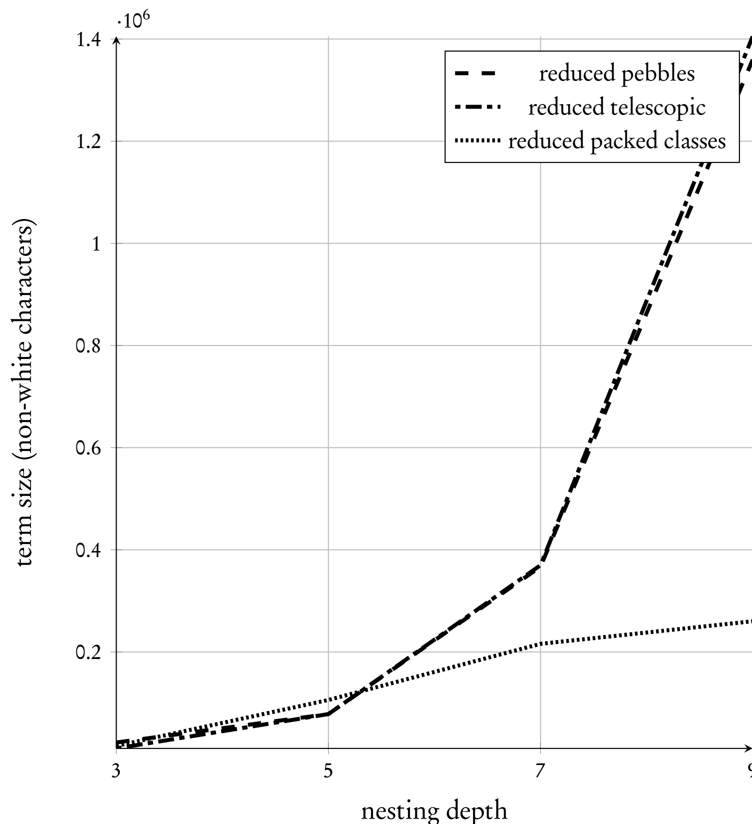


Figure 1.30: Size of the ordered type structure term on lists (of lists of ...) booleans, according to each paradigm.

We have found that the packed classes discipline deals coherently with the pitfalls of the telescopic-style design we mentioned:

- The key that triggers the inference of a given structure is the first projection of that structure. It directly projects to the object (type or value) we might want to infer the structure on. In case of a `Type` structure, this means the carrier type, for instance. This implies that the inference of a structure arbitrarily high in a refinement chain is done in a single `Canonical Structure` search.
- When refining a type with distinct, non-mutually-dependent mixins, nothing forces us to encapsulate one inside the other any more. Said in another way, for any structure, the set of its parents is usually mentioned in its *class*, in positions that are roughly equivalent, and more importantly, in a context independent from inference. In the case where this set of parents is larger than a singleton, nothing prevents us from declaring `Canonical` hints for **all** parents, re-composed from hand-crafted `Structure` projections. We can thus model multiple inheritance easily. We explain how in detail, in § 1.4.2 on p. 67.

Finally, the `class_of` record permits us to mention only once the mixins of diverse refinement rank composing a refined `Structure`: the dependent record lets us specify parametric members depending the ones on the others, and thus internalizes references to other members of the inheritance graph. This means concretely, that from the point of view of inheritance, if n is the refinement depth, the size of the record the user has to specify is back to C^n, with $C = 1$ — as with telescopes.

This is not the case for a nested record, however, at least in the frequent case where a parametric nesting has a concrete dependency on a nested operation defined at the $n-1$ nesting level, as is the case with the order in Fig. 1.31 on p. 66 : each member of the equality type on lists, as well as each member of the ordered type on lists, has to carry a copy of the underlying equality (resp. order) type on their underlying parameter. But the advantage of packed classes is that the equality and order mixins are accessible in tight, separated bundles defined as independent values *for distinct inductive types*.

To verify this effect, we have measured the size of records fully reduced to head normal form[71] using our example of an ordered type instance for lists of booleans, at increasing levels of nesting. The crucial generic instance declaration is as such:

[71] We will give more elements about the specific importance of the size in fully reduced head normal form in § 1.4.4 on p. 74.

```
Definition list_eqType (eT:eqType) :=
  EqType (list eT) (EqMixin _ (@eql eT)).

Definition list_ordType (oT : orderedType):=
  OrderedType (list oT)
  (OrderedMixin _ _ (@ord_antiH oT) (@ord_transitH oT)).
```

The user definitions of the nested instances involved are identical to the letter to those of the telescopic style featured in § 1.1.12 on p. 34 (a boon of the strict naming and notation patterns). We used the COQ pretty-printer to measure size, because it has two particular properties:

- it increases indentation considerably after each δ-expansion of a type constructor,
- and it cuts printing (replacing the "deep" code fragment by "..") after a certain level of indentation

This allows us to measure cheaply the depth of folding of our term under inductive constructors. As we have noticed in § 1.2.4 on p. 42, this is a pertinent measure because unification procedures delay this unfolding. Meanwhile, the storage of inductive instances in a partitioned environment (see Tab. 1.2 on p. 23(b)) means every inductive definition in a term incidentally allows term comparison procedures to access an explicit synchronization construct.

Fig. 1.30 on the preceding page shows the size of records in number of non-blank characters, as declared in the three paradigms, from the ordered `Structure` of a list of lists of booleans, to that of a "list of lists of lists of lists of lists of lists of lists of booleans". It is clear that the growth of the record term in both parametric and telescopic paradigms is exponential, and that packed classes, under the hypothesis of careful folding of inductive instances, allow to keep it "linear". However, with the use of implicit arguments, and by avoiding full reduction when displaying terms to the user, this is not apparent to the average COQ user.

Thus, more than anything else, the study of term size implied by

record paradigms as a function of refinement and nesting depths shows that it requires complex libraries dealing with deeply-imbricated structures to detect the critical performance issues related to those models.

paradigm	term size as function of	
	inheritance level	nesting level
Pebbles	exponential	exponential
Telescopic	linear	exponential
Packed	linear	δ-folded exponential

Table 1.5: Growth of record size, according to each paradigm

```
Definition rel T :=
  (T → T → bool).

Module Equality.

Record mixin_of (T: Type) :=
  Mixin { op : rel T }.
Notation class_of :=
  mixin_of (only parsing).

Section ClassDef.
Structure type := Pack {
  sort;
  _ : class_of sort;
  _ : Type
}.
Local Coercion sort :
  type ↦ Sortclass.

Variables (T: Type) (cT: type).
Definition class :=
  let: Pack _ c _ as cT' := cT
  return class_of cT' in c.
Definition pack c := @Pack T c T.
Definition clone :=
  fun c & cT → T & phant_id (pack
     c) cT
  = > pack c.
End ClassDef.

Module Exports.
Coercion sort : type ↦ Sortclass.
Notation eqType := type.
Notation EqMixin := Mixin.
Notation EqType T m := (@pack T m).
End Exports.

End Equality.
Export Equality.Exports.

Definition eq_op (T: eqType) :=
  Equality.op T (Equality.class T).
```

```
Definition asymm (eT: eqType)
  (o: eT → eT → bool) : Prop :=
  ∀ (x y : eT), (o x y) →
    (o y x) → (eq_op _ x y).

Definition transit (T: Type)
  (o: T → T → bool) : Prop :=
  ∀ (x y z : T),
  (o x y) → (o y z) → (o x z).

Module Ordered.

(* asymm depends on eq_op *)
Record mixin_of (T: eqType) :=
  Mixin {
    ord  : T → T → bool;
    _ : asymm _ ord;
    _ : transit _ ord
  }.

Section ClassDef.
Record class_of T := Class {
  base : Equality.class_of T;
  mixin : mixin_of (EqType T base)
}.
Local Coercion base :
  class_of ↦ Equality.class_of.

Structure type := Pack {
  sort;
  _ : class_of sort;
  _ : Type
}.
Local Coercion sort :
  type ↦ Sortclass.

Variables (T : Type) (cT : type).
Definition class :=
  let: Pack _ c _ as cT' := cT
  return class_of cT' in c.
Definition clone c of phant_id
     class c :=
  @Pack T c T.
Definition pack b0 (m0: mixin_of
     (@Equality.Pack T b0 T)) :=
  fun bT b & phant_id
     (Equality.class bT) b = >
    fun m & phant_id m0 m = > Pack
     _ (@Class T b m) T.
(* Inheritance *)
Definition eqType := Equality.Pack
     _ class cT.
End ClassDef.

Module Exports.
Coercion base : class_of ↦
     Equality.class_of.
Coercion sort : type ↦ Sortclass.
Coercion eqType : type ↦
     Equality.type.
(* Inheritance *)
Canonical Structure eqType.
Notation orderedType := type.
Notation OrderedMixin := Mixin.
Notation OrderedType T m := (@pack
     T _ m _ _ id _ id).
End Exports.

End Ordered.
Export Ordered.Exports.

Definition ord T :=
  Ordered.ord _ (Ordered.mixin _
     (Ordered.class T)).
```

```
Module Lattice.

(* The property depends on ord*)
Record mixin_of (O: orderedType) :=
  Mixin {
    bottom : O;
    top : O;
    meet : O → O → O;
    join : O → O → O;
    _ : ∀ x y, (ord _ x y) ↔
      (eq_op _ (join x y) y)
  }.

Section ClassDef.
Record class_of T := Class {
  base : Ordered.class_of T;
  mixin : mixin_of (Ordered.Pack _
     base T)
}.
Local Coercion base : class_of ↦
     Ordered.class_of.

Structure type := Pack {
  sort;
  _ : class_of sort;
  _ : Type
}.
Local Coercion sort : type ↦
     Sortclass.

Variables (T : Type) (cT : type).
Definition class :=
  let: Pack _ c _ as cT' := cT
  return class_of cT' in c.
Definition clone c of phant_id
     class c :=
  @Pack T c T.
Definition pack b0 (m0: mixin_of
     (@Ordered.Pack T b0 T)) :=
  fun bT b & phant_id
     (Ordered.class bT) b = >
    fun m & phant_id m0 m = > Pack
     _ (@Class T b m) T.
(* Inheritance *)
Definition eqType := Equality.Pack
     _ class cT.
Definition orderedType :=
     Ordered.Pack _ (base _ class)
     cT.
End ClassDef.

Module Exports.
Coercion base : class_of ↦
     Ordered.class_of.
Coercion mixin : class_of ↦
     mixin_of.
Coercion sort : type ↦ Sortclass.
Coercion eqType : type ↦
     Equality.type.
(* Inheritance *)
Canonical Structure eqType.
Coercion orderedType : type ↦
     Ordered.type.
(* Inheritance *)
Canonical Structure orderedType.
Notation latticeType := type.
Notation LatticeType T m := (@pack
     T _ m _ _ id _ id).
Notation LatticeMixin := Mixin.
End Exports.

End Lattice.
Export Lattice.Exports.
```

Figure 1.31: A well-tiered packed-classes-style decidable lattice *concept*-ualization

1.4.2 *Multiple Inheritance with packed classes*

Complex *vertical* sharing is easily expressed with the packed classes paradigm: indeed, vertical sharing is about joining distinct refinements of common ancestors, and the flexibility with which we can express with parametric mixins, joined with the automatic handling permitted by a bundled representation, provides an expressive result.

As a complex example of multiple inheritance — for which we want to elicit a comparison with Spitters and van der Weegen (2011) — we consider an equivalence relation. We would like to define such a refined relation by inheritance from the *reflexive*, *symmetric*, and *transitive* relation types definable in such a case.

The first issue of this construction is *sharing*: we want to ensure that those three objects will refer to the same relation and the same carrier type. It is natural that we will have to require three "reflexive", "symmetric", and "transitive" mixins in the *class* of the final equivalence relation. In our paradigm, those mixins will need to take the base type we want them to share as a parameter. We therefore give the defining elements of a bundle for that synchronization unit in Fig. 1.32: it is in itself but a type with an additional operation, a textbook use case of our notion of mathematical structure. We denote it outside of its module using our usual convention, with the prefix `rel/Rel`:

- `relType` denotes a *type with relation* structure,
- `RelMixin` is the constructor of the *relation mixin*,
- `RelType` is the constructor of the *type with relation* structure, taking a type and a *relation mixin* for that type as arguments.

As usual, we also define our usual auxiliary functions, notably `class`, which, given a *type with relation* instance, returns the `class_of` record used to form it, and `pack`, which given a relation mixin, forms the inferrable record instance that can then be declared `Canonical`. All this is systematic in our class definitions, and as has been hinted at in the last subsection, we can be content with specifying a structure using its name and definitions in the like of Fig. 1.32.[72] We also provide a projector `rel` to access the relation operator buried in the mixin of our type with relation structure more easily:

```
Definition rel (rT:relType) := Relation.rel (Relation.class rT).
```

We then extend that bundle with each of the reflexive, symmetric or transitive property. We consider only the reflexive relation `rrelType` in Fig. 1.33 on the following page: the other definitions `srelType` and `trelType`, respectively for the symmetric and transitive relations are structurally identical. It consists simply in defining a single-property refinement, for which the SSReflect idiom would usually be to use telescopes, especially for hierarchies of moderate depth: the telescopic extension is, with one single member added to its parent, of the same size as the mixin-based extension offered by packed classes.

However, for this example, we are going to consider that we would normally plan to refine relations considerably beyond the *equivalence relation* stage, and play the structural game fully. In that case, the definitions of Fig. 1.33 on the next page are supplemented by the usual constructors (`pack`), destructors (`class`) and type abbreviations (`rrelType`, ...). To those we add

[72] The exact mechanism for `pack` and `class` in the most complex cases will be touched upon but in 2.3.2 on p. 111. We hope the purpose of this construct — if not the *how* — will remain clear nonetheless.

```
Module Relation.
Record mixin_of (T:Type) :=
  Mixin { rel : relation T }.
Notation class_of :=
  mixin_of (only parsing).
<...>
End Relation.
```

Figure 1.32: The `relType` relation structure

```
Module ReflexiveRelation.
Record mixin_of (rT:relType) := Mixin {
  _ : reflexive _ (rel rT)
}.
Record class_of (T:Type) := Class {
  base : Relation.class_of T;
  mixin : mixin_of (Relation.pack base)
}.
<...>
End ReflexiveRelation.
```

Figure 1.33: `rrelType`, a reflexive relation structure (denoted with prefix `rrel`)

the inheritance-inducing `Canonical Structure` declaration, that we note here as a further reminder of the inheritance paradigm exposed in § 1.4.1 on p. 60. Let us therefore look at another subset of the declaration of the `ReflexiveRelation` module, elided (with "...") in Fig. 1.33:

```
Module ReflexiveRelation.
<...>
Section ClassDef.
Structure type := Pack { ... }.
Definition class : ∀(cT:type), class_of (sort cT) := ...
<...>
Definition relType := Relation.pack class.
End Section ClassDef.

Module Import Exports.
Coercion relType : type ↦ Relation.type.
Canonical Structure relType.
<...>
End Exports.
End ReflexiveRelation.

Import ReflexiveRelation.Exports.
```

The multiple inheritance necessary to define an equivalence relation then comes at a fairly cheap price: it suffices to unite three reflexivity, symmetry, and transitivity mixins on a given, common relation type. We do that directly in a `class_of` mixin: since there is no proper equivalence property to add, the definition of Fig. 1.34 on the facing page is purely structural.

This definition provides the easy synchronization of the three mixins on a common parent, while also providing the flexibility of having separate reflexive, symmetric and transitive relation instances for each equivalence relation instance : **the packed classes design preserves the inferrability of the telescopic design, while subsuming the modularity of the pebble-style design**. Concluding on the example proposed by Spitters and van der Weegen (2011, §3), we can say that there is no inherent synchronization issue with bundling. The real issue, duly noted by Spitters and van der Weegen, is that in Coq, we can only define a single instance of equivalence relation for a given inference-directing object (in our case the first projection to `Type`). While it would certainly be an improvement to have more flexibility in that regard,[73] this issue should be taken up as a limitation of the particular `Canonical Structure` inference mechanism as implemented in Coq, and not as an inherent failure of the structuring mechanism that bundling represents. Thanks to the proofs already contained in the module `Equalities` in the standard library, we can now provide an example of the instances declaration for our structures, that of equality. It is represented in Fig. 1.35 on the facing page. As an added bonus, we use in its last line a

[73] We will also address ways to extend on that limitation with `Canonical Structure`s in § 1.4.4 on p. 74.

```
Module EquivalenceRelation.

Section ClassDef.
Record class_of (T : Type) : Type := Class {
  base : Relation.class_of T;
  mixin1 : ReflexiveRelation.mixin_of (Relation.pack
     base);
  mixin2 : TransitiveRelation.mixin_of (Relation.pack
     base);
  mixin3 : SymmetricRelation.mixin_of (Relation.pack
     base)
}.

Local Coercion base : class_of ↦ Relation.class_of.
Definition base1 R m := ReflexiveRelation.Class
     (@mixin1 R m).
Local Coercion base1 : class_of ↦
     ReflexiveRelation.class_of.
(* similar definitions base2, base3 for
   SymmetricRelation, TransitiveRelation *)

Structure type := Pack {sort; _ : class_of sort; _ :
     Type}.
Local Coercion sort : type ↦ Sortclass.
Variables (T : Type) (cT : type).
Definition class :=
  let: Pack _ c _ as cT' := cT return class_of cT' in c.
Definition pack := ...

Definition relType := Relation.Pack class cT.
Definition rrelType := ReflexiveRelation.Pack class cT.
(* similar definitions srelType, trelType *)
End ClassDef.

Module Import Exports.
Coercion base : class_of ↦ Relation.class_of.
Coercion mixin1 : class_of ↦
     ReflexiveRelation.mixin_of.
Coercion base1 : class_of ↦
     ReflexiveRelation.class_of.
(* similar Coercion declarations mixin2, base2,
   mixin3, base3 *)

Coercion sort : type ↦ Sortclass.
Coercion relType : type ↦ Relation.type.
Canonical Structure relType.
Coercion rrelType : type ↦ ReflexiveRelation.type.
Canonical Structure rrelType.
(* similar Coercion and Canonical Structure declarations
   srelType, trelType *)
Notation eqrelType := type.
Notation EqRelType T := ...
End Exports.

End EquivalenceRelation.
Import EquivalenceRelation.Exports.
```

Figure 1.34: `erelType`, an equivalence relation structure (denoted with prefix `erel`)

```
Require Import Equalities.

Definition eq_relmixin T := (RelMixin (@eq T)).
Canonical Structure erel T := Eval hnf in RelType _ (eq_relmixin T).

Definition eq_rrelmixin T := (RRelMixin (@eq_refl T)).
Canonical Structure errel T := Eval hnf in RRelType _ (eq_rrelmixin T).

Definition eq_srelmixin T := (SRelMixin (@eq_sym T)).
Canonical Structure esrel T := Eval hnf in SRelType _ (eq_srelmixin T).

Definition eq_trelmixin T := (TRelMixin (@eq_trans T)).
Canonical Structure etrel T := Eval hnf in TRelType _ (eq_trelmixin T).

Canonical Structure eerel T := [eqRelType of T].
```

Figure 1.35: The reflexive, symmetric, transitive and equivalence relation instances for equality.)

notational tool that we have devised to refer to inferred parent structures implicitly, but that we will only expose in § 2.3.2 on p. 111. This makes the instance declarations for our structures relatively lightweight, but the user might want to consider this line replaced by an explicit reference to the `pack` constructor followed by passing the instances `eq_rrelmixin`, `eq_srelmixin` and `eq_trelmixin` in the meanwhile.

1.4.3 Manifest-like records and multiple dispatch

Complex synchronization with multiple parameters Packed classes are more expressive than telescopic-style records for the *horizontal* sharing of type information between concepts, giving us enough flexibility to represent equational identities between members of a `Structure` through the parameters of said `Structure`.

As mentioned in § 1.1.10 on p. 32, modelling type-level constraints, such as those involved in the definition of a categorical adjunction, can be problematic without the boon of either type parameters or manifest records. However, since we de-couple `Structure` inference from type parameters of a record, it is possible to use those parameters in a indicative fashion, to expose more information about the elements of this record. That remark was already made in a more abstract manner at the end of § 1.3.2 on p. 53. However, with packed classes, we can also integrate those *indicative, rather than structural* parameters with a coherent inheritance mechanism. This allows us to build adjunctions without problems — let us see how.

Since we only want this to serve for a toy example,[74] the definition of a category, inferrable from a type, is straightforward (Fig. 1.36) from the mathematical definition : a category $\mathcal{C}$ is given by a type of objects $\mathrm{Op}(\mathcal{C})$ and type of arrows, mappings between elements of $\mathrm{Ob}(\mathcal{C})$ equipped with an identity and a composition. The prerequisite `left_id`, etc, property definitions are in `ssrfun` in the SSReflect library for the curious reader to consult.

The key part in our example is the definition of a functor: a functor is a pair of mappings between a *source* and a *target* category:

- an *object map* from source to target objects,
- and an *arrow map* from source to target arrows.

The arrow map has to associate the identity element of the target to that of the source, and be continuous with respect to source and target compositions. The object map has to agree with the arrow map in one of two possible ways depending on whether we deal with a *covariant* or *contravariant* functor:

- the image of an arrow from object A to B is mapped to an arrow going from the image of A to that of B for a covariant functor,
- or to an arrow going from the image of B to that of A for a contravariant functor.

In this toy example, we will only deal with covariant functors. The key obstacle with the definition of an adjunction — which, let us remind, is formed using two functors linking the same categories, but in opposite directions — is that if the type of a functor does not reflect its source and target categories, it is very cumbersome to verify the adjunction constructor

```
Record mixin_of (oT: Type) :=
    Mixin {
  composition : (oT → oT) →
    (oT → oT) → (oT → oT);
  _ : associative composition;
  identity : oT → oT;
  _ : left_id identity
    composition;
  _ : right_id identity
    composition
}.

Notation class_of :=
  mixin_of (only parsing).
```

Figure 1.36: The mixin giving rise to `catType`, a Category structure pegged on a `Type`. We are simplifying the definition of a category greatly, here — but it doesn't impact on the point we are trying to make about sharing.

[74] Note that since we have no overlapping instances, this example is limited to a single instance of category per Coq type, which prevents Coq from inferring the e.g. dual category $\mathcal{C}^{op}$ of an existing instance, because it shares the same carrier type. Should we have wanted to make this example scale, and to keep such a low-level (universe-wise) definition, we might have turned to something like the indexation of the structure on the *arrow* type found in Haskell's `Data.Category`

is being passed appropriate functors:

```
Definition adjunction (F : Functor) (G : Functor),
  src F = dst G → dst F = src G → ...
```

As mentioned by Sozeau and Oury (2008):

> This gets very awkward because the equalities will be needed to go from one type to another in the definitions, obfuscating the term with coercions, and the user will also have to pass these equalities explicitly.

Sozeau and Oury continue showing that, with the discipline of having superclasses as parameters, it is possible to specify adjunctions because of a *parametric* definition of functors : the functor type carries its source and target. We notice that with packed classes, it is possible to replicate such an exposition of superclass types as parameters, but on a voluntary basis.

The object and arrow mappings are straightforward (Fig. 1.37). As explained in § 1.3.4 on p. 56, even if a functor only makes sense between categories, we define them on the carrier type of the category structure: this lets COQ insert the inference-inducing record projections in our term by coercion. We then define the appropriate mixin based on the categories required on the source and target type (Fig 1.38) — indeed, without a category structure on both source (`sT`) and target (`tT`) of the functor's mapping function, the correspondence between the image of the source's identity and the identity of the target (`func_id`) can't be expressed, forcing the *functor mixin* to depend on category types.

The functor is an object essentially defined by its mappings, so that it seems natural to peg it on a COQ function, for which we arbitrarily choose the object map, for simplicity. This means that we will want a functor instance inferred at a time where we are involved in the concrete manipulation of something corresponding to its object map, and leads to the `map` structure defined in Fig. 1.38. The change of naming pattern reflects that we do not have a simple type as a first argument. This choice of first projection can be adapted to fit a use case, letting either or both mappings be canonical projections of the functor structure.

```
Implicit Types (sT tT : Type).

Definition object_map sT tT :=
  sT → tT.
Definition arrow_map sT tT :=
  (sT → sT) → (tT → tT).

Definition covariant_map sT dT
  (Fo: object_map sT dT)
  (Fa: arrow_map sT dT) :=
  ∀(f : sT → sT) (x: sT),
    Fo (f (x)) = Fa(f) (Fo(x)).

Implicit Types (cT dT : catType).

Definition func_id cT dT (Fa :
    arrow_map cT dT) :=
  Fa(@identity cT) = (@identity
    dT).
Definition func_comp cT dT
    (Fa: arrow_map cT dT) :=
  ∀(f g : cT → cT),
  Fa (composition g f) =
    (composition (Fa g) (Fa f)).
```

Figure 1.37: The base mappings and property definitions forming the foundation for a functor

```
Variables (cT dT: catType).

Record mixin_of (oM : object_map cT dT) := Mixin
    {
  aM : arrow_map cT dT;
  _ : covariant_map oM aM;
  _ : func_id aM;
  _ : func_comp aM
}.

Notation class_of := mixin_of.

Structure map (phcd: phant (cT → dT)) := Pack {
  apply : cT → dT;
  _ : class_of apply;
  _ : cT → dT
}.

Local Coercion apply :
  map ↦ Funclass.
```

```
Variables (phCD : phant (cT→ dT))
  (f g : cT → dT) (cF : map phCD).
Definition class :=
  let: Pack _ c _ as cF' := cF
  return class_of cF' in c.
Definition clone fA of phant_id g (apply cF)
  & phant_id fA class := @Pack phCD f fA f.
Definition pack (fZ : mixin_of f) :=
 @Pack (Phant (cT → dT)) f fZ f.

End ClassDef.

Module Exports.
Coercion apply : map ↦ Funclass.
Notation Functor fCD := (@pack _ _ _ fCD).
Notation "{ 'functor' fCD }" := (map (Phant fCD))
  (at level 0, format "{ 'functor'  fCD }").
Notation "[ 'functor' 'of' f 'as' g ]" :=
    (@clone _ _ _ f g _ _ idfun id)
  (at level 0, format "[ 'functor'  'of'  f  '
    as'  g ]") : form_scope.
```

Figure 1.38: The mixin & base structure of a functor, based on `catType` instances for source & target. (customarily included in a `Module Functor`).

The key definition is in the *notations* that follow the functor definition

and allow us to require explicit naming of the usually-implicit category parameters of the functor structure. The SSReflect approach to structure parameters is to systematically use such a notation to develop functor theory — giving this type to abstract variables — as well as to define other structures depending on a functor. The definition of `apply` (which, if coercions are made explicit, unfolds to `Functor.sort cT → Functor.sort dT`) ensures that the functor is inferrable simply from the occurrence of its object mapping in any expression.

The definition of a functor would have been equally workable with a definition such as the following:

```
Structure map := Pack { apply : cT → dT;
       _ : class_of apply; _ : cT → dT }.

<...>
Notation "{ 'functor' fC ~> fD }" := (@map fC fD)
  (at level 0, format "{ 'functor'  fC ~> fD }").
```

But, while we wanted to treat the example suggested by Sozeau and Oury (2008), we wanted to show the technology that allows us to reflect complex types in notations, possibly allowing a library architect to choose as indicative structure parameters identifiers that for some reason come *late* in the prenex order of the large dependent product that is our structure. Indeed, we noticed that in telescopic style, the element that needs to be a projection of the inferrable record is the COQ function `(cT → dT)`, and not necessarily the source and target category types. Since the difficulty with reconciling adjunctions and `Canonical Structures` is *the perception that this first projection can't be used as a parameter* to assess the exact nature of functors from their type, using a type parameter based on anything else that this exact mapping `(cT → dT)` would have felt like a cop-out.

To that aim, reusing the native COQ arrow type (_ → _) in our notation, we use a phantom definition, in effect a singleton type decoration — or equivalently, a unit type with a mandatory type parameter

We will come back to phantoms in 2.3.2 on p. 108, and for now, we ask the user to consider that this simply allows us to give a compact single-argument specification for our arrow type in the`{functor ...}` notation.

From that point, the definition of base instances such as identity functor ($1_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$, Fig. 1.39) or the functor rising from the composition of two functors, and the definition of natural transformations[75] present no particular difficulties. They allow us to get to the definition of an adjunction.

[75] A *natural transformation* η from functor F to functor G, both sharing the same source and target categories $\mathcal{C}$ and $\mathcal{D}$, associates to each object X of $\mathcal{C}$ a mapping η_X relating F(X) to G(X) called the *component* of η in X, so that $\forall f : X \mapsto Y$ arrow of $\mathcal{C}, \eta_Y \circ F(f) = G(f) \circ \eta_x$. The full code for the definition of a natural transformation, as well as the full development of this category-theoretic subsection, are available as developed from a stock SSReflect release 1.3pl1, on Github: https://github.com/huitseeker/thesis-spikes/tree/Adjunctions

```
Definition idmap T := @id T.
Section IdentityFunctor.

Variable cT : catType.

Notation idmap := (@idmap cT).

Lemma id_covariance :
  @covariant_map _ cT (idmap) id.
Proof. by []. Qed.

Lemma id_fidentity :
  @func_id cT cT id.
Proof. by []. Qed.

Lemma id_fcomposition :
  @func_comp cT cT id.
Proof. by []. Qed.

Canonical identityFunctor := Functor
  (Functor.Mixin
     id_covariance
     id_fidentity
     id_fcomposition).

End IdentityFunctor.
```

Figure 1.39: The definition of the identity functor.

An adjunction between categories $\mathcal{C}$ and $\mathcal{D}$ is defined — among other equivalent definitions — as a tuple (F, G, η, ϵ), where:

$$F : \mathcal{C} \to \mathcal{D} \text{ is a functor} \qquad \eta : 1_{\mathcal{C}} \to GF \text{ is a natural transformation}$$
$$G : \mathcal{D} \to \mathcal{C} \text{ is a functor} \qquad \epsilon : FG \to 1_{\mathcal{D}} \text{ is a natural transformation}$$
$$\begin{cases} (G\epsilon) \circ (\eta G) = 1_G \\ (\epsilon F) \circ (F\eta) = 1_F \end{cases}$$

The use of two functors in that definition forces us to consider which object should be the canonical projection of an adjunction. Since the definition is symmetrical, we choose one of the functors. The second functor

(the adjunct) will occur as a simple parameter of the structure, to be passed as an argument during canonical instance registration: in particular, the adjunction will not coerce to it so as not to create an incoherent path.

In packed classes style, this will require us to have a parent mixin (here, a `Functor.mixin_of`) as the first member of a `class`, while a proper child `mixin_of` will define the specific properties required of an adjunction. **The final definition for an adjunction parametrized by two functors with chiasmatic source and target constraints is featured in Fig. 1.40.** Our structure is inferrable, despite the fact that the first projection of our functor is a mapping between category types — the concrete COQ object the user will enjoy manipulating and inferring an `{adjunction cT → dT}` on —, not a functor. We nonetheless managed to make the "child" adjunction mixin correspond exactly to what would have been specified in pebble-style. Though the parametricity of the adjunction structure in the *adjunct* functor isn't strictly necessary, we hope it shows off the flexibility of the packed classes idiom.

We leave the definition of a category structure which supports duality (see note 74 on p. 70), and the subsequent definition of the dual adjunction structure as an exercise to the reader.

```
Variables (cT dT:catType).

Record mixin_of (F : {functor cT → dT})
  (G: {functor dT → cT}) :=
  Mixin {
    unit : {nattrans (@idmap cT) ~> (G\o F)};
    counit : {nattrans (F\o G) ~> (@idmap dT)};
    _ : ∀Y : dT,
      ((armap G \o counit) Y) \o
              ((unit \o G) Y) = (@idmap cT);
    _ : ∀X : cT,
      ((counit \o F) X) \o
               ((armap F \o unit) X) = (@idmap
    dT)
  }.

Record class_of f (G: {functor dT → cT}) :=
    Class {
  base : Functor.mixin_of f;
  mixin : mixin_of (Functor base) G
}.
```

```
Structure map (phCD : phant (cT → dT)) := Pack {
  apply : cT → dT;
  unapply : {functor dT → cT};
  _ : class_of apply unapply;
  _ : cT → dT
}.

Local Coercion apply :
  map ↦ Funclass.

Variables (phCD : phant (cT → dT))
  (f : cT → dT) (G : {functor dT → cT}).
Variables  (cF : map phCD).
Definition pack F G (m : mixin_of F G) :=
  fun b & phant_id (Functor.class F) b =>
    fun m0 & phant_id m0 m =>
      Pack phCD (@Class f G b m0) f.

Notation Adjunction m :=
  (@pack _ _ _ _ _ _ m  _ id _ id).
Notation "{ 'adjunction' fCD }" :=
  (@map _ _ (Phant fCD))
  (at level 0, format "{ 'adjunction'  fCD }").
```

Figure 1.40: The definition of an adjunction — *i.e.* the contents of the `Adjunction` module — with type-level functor compatibility verification. (`armap` is the arrow map of a functor)

Multiple Dispatch Type class systems found their origin in the desire for a type semantics providing an equivalent to the notion of method overloading. As such, the question of how to extend them in directions parallel to those followed by the concept of objects method selection makes sense. Multiple dispatch, the selection of a method depending on several objects at once, is one of those extensions.

How do we select an implementation based on multiple values ? If we know all the possible data variants for each of the values, it's a relatively easy problem: we make each variant an alternative of a data type, and pattern-match on both alternatives. We show in Fig. 1.41 on the next page how that looks like in COQ.[76]

[76] As usual in SSReflect parlance, the `CoInductive` keyword is just here to avoid the generation of recursors (Gonthier et al. 2008, §11.1).

```
CoInductive Shape :=
  | Rectangle : ∀(x y z t: nat), Shape
  | Circle : ∀(x y r: nat), Shape.

Definition intersection : Shape → Shape → bool.
refine (fun s1 s2 => match s1 with
 | Rectangle x1 y1 z1 t1 => match s2 with
          | Rectangle x2 y2 z2 t2 => _
          | Circle x2 y2 r2 => _
        end
 | Circle x1 y1 r1 => match s2 with
          | Rectangle x2 y2 z2 t2 => _
          | Circle x2 y2 r2 => _
        end
end); admit.
Defined.
```

Figure 1.41: Datatype-based multiple dispatch for closed types

In systems with a native implementation of multiple parameter classes, the goal is to leave some room for the further extension of that solution : the programmer generally wants to define more and more specialized instances (in our examples, cases for each shape), so we aim at leaving room for a more efficient implementation than the previous alternatives. The gist of this extensible idiom is to declare each data variant (`Rectangle`, `Circle`) as an instance of a `Shape` `Structure`, and to have another `Structure` specific to each multi-method that has to be implemented over said `Shape`(s).

In our example, that means having an `Intersection` `Structure` that inherits from two `Shape`s, and implements the `intersection` method (of a specified type) accordingly.

How do we obtain distinct instances of `Intersection` for the four possible `Shape` instances passed as an arguments ?

As a first approach, and as far as the `Canonical Structure`s mechanism is concerned, we do not. The extension of unification we rely on fixes a particular form to the disagreement pair we can work on (§ 1.2.2 on p. 37), and this one features but a single meta-variable for a projection value. However, there are common approaches to encoding multiple dispatch while using only single dispatch, made famous by the long-standing restriction of popular languages like Java or C# to the latter.

Using `Canonical Structure`s, we have seen that we can resort to some of them using Packed Classes (§ 1.4.2 on p. 67) : our pattern for multiple inheritance implements exactly the emulation of multiple dispatch using several single-dispatch messages.

1.4.4 *Overlapping instances, instance chains, and Data types à la carte*

Overlapping instances The expression problem[77] has a well-known type-class based solution (Lämmel and Ostermann 2006, §2.4). The standard example for this benchmark of extensibility in both the dimensions of *data* and *operations* is an arithmetic evaluator. We present the `Canonical Structure` syntax of the classic solution in Fig. 1.42 on the next page, in telescopic style to maximize brevity. It models each data variant of the `Expr` data type as an instance of an overruling *class* (`Structure`), and registers a specific type as an instance for each data variant. `Expr` refers to the extensible, nominal union

[77] That we introduced in § 1.3.3 on p. 54.

of all data variants, and is deferred to when recursing over each subterm of our expressions (the equivalent of a trampoline in tail call elimination). Each extensible operation in the `Expr` "datatype" is modelled as a *refinement*, that adds implementations of said operation for each variant as instances. We recognize in this example a single-inheritance version of the type-class based multiple dispatch implementation of the previous section.

The complete analysis of that solution is found in the literature (*ibid.*), but the gist of it is that, while ensuring type safety across extensions in both dimensions, this solution forces the user to adhere to a specific pre-existing style. That this style occurs particularly often in the SSReflect libraries is intriguing, but anecdotal: the expression problem is posed in the more general framework of generic programming. And it is to solve that general problem that a solution based on Haskell's *overlapping instances* was proposed (Swierstra 2008).

Two `Canonical Structure` (or type class) instances overlap if they could apply to the same value. For example, consider the following `Structure`:

```
Structure c := C {
  fst :> Type;
  snd : Type
}.

Canonical Structure ins1 := C nat nat.
Canonical Structure ins2 (A:Type) := C nat A.
```

Both instances could match a given natural. However, the compiler has no guarantee that the members of the corresponding records are implemented equivalently for both, so that a proof using both instances would have several interpretations. To eliminate the non-determinism, Haskell 98 forbids overlapping instances.

Under some conditions, there are less restrictive alternatives: Scala, for example, leverages its object model to disambiguate between implicits using subclassing (Oliveira et al. 2010, §6.5). The GHC implementation of Haskell, given `-fallow-overlapping-instances` , will allow overlapping instances, choosing the most specific in a precise sense (Peyton-Jones et al. 1997): an instance `a` is more specific than `b` if `a` is a substitution instance of `b`, but the converse isn't true. In our example above, the first instance would have been chosen according to that rule.

Swierstra solves the expression problem by defining expressions using ground types and a (categorical) coproduct constructor `t :+: u`:

```
data (f :+ : g) e = Inl (f e) | Inr (g e)
```

To state that there exists an injection from `(f e)` to `(g e)`, he defines a subtyping relation `f :≺: g`

```
class f :≺: g where inj :: f e → g e
```

Then he fills this class with the following three overlapping instance declarations:

```
instance f :≺: f
instance f :≺: (f :+ : g)
instance f :≺: h => f :≺: (g :+ : h)
```

The *substitution rule* for instance selection constrains the use of those expressions: because the second instance is a substitution instance of the last,

```
Structure exprType := Expr {
  sort :> Type
}.
CoInductive Lit :=
 | in_lit (proj_lit : nat) : Lit.
Definition proj_lit (x:Lit) :=
  let: in_lit k := x in k.
CoInductive Addit
  (x y : exprType): Type :=
  | in_addit (a: x) (b: y): Addit x y.
Canonical Structure lit_expr :=
  Expr Lit.
Canonical Structure addit_expr
  (x y:exprType) :=
  Expr (Addit x y).

Structure evalType := Evaluator {
  evaluatee :> exprType;
  eval : evaluatee → nat
}.
Canonical Structure lit_eval :=
  Evaluator proj_lit.
Definition eval_addition
  (x y:evalType) := fun z = >
   let: in_addit a b := z in
     (@eval x a) + (@eval y b).
Canonical Structure addit_eval
  (x y:evalType) :=
  Evaluator (@eval_addition x y).

(* Extension in data *)
CoInductive Pred (x:exprType) :=
| in_pred (a:x) : Pred x.
Canonical Structure pred_expr
  (x:exprType) := Expr (Pred x).
Definition eval_pred (x: evalType) :=
  fun z = > let: in_pred a:= z in
    (@eval x a).−1.

(* Extension in operation *)
Require Import Ascii String.

Structure printType := Printer {
  printee:> exprType;
  print: printee → string
}.
(* Left as an exercise to the reader *)
Definition string_of_nat : nat →
     string.
admit. Defined.

Definition print_lit :=
  fun x = > string_of_nat (proj_lit x).
Canonical Structure printer_lit :=
  Printer print_lit.

Definition print_add
  (x y : printType) := fun z = >
  let: in_addit a b := z in
  (append (append (@print x a) "+ ")
     (@print x b)).

Definition print_pred (x:printType) :=
  fun z = > let: in_pred a := z in
    (append (@print x a) "−1").

Canonical Structure printer_addit
  (x y :printType) :=
  Printer (@print_add x y).

Canonical Structure printer_pred
  (x:printType) :=
  Printer (@print_pred x).
```

Figure 1.42: The classical type-class-based solution to the expression problem

predicates will only be checked against the last rule if they fail to match the second — making the coproduct constructor behave like a list constructor, rather than like a tree constructor.

> We use coproducts in a list-like fashion: the third instance only searches through the right-hand side of coproduct. Although this simplifies the search—we never perform any backtracking—it may fail to find an injection, even if one does exist. For example, the following constraint will not be satisfied:
>
> ```
> f :≺: ( (f :+ : g) :+ : h )
> ```
>
> Yet clearly `Inl ;o; Inl` would be a suitable candidate injection. Users should never encounter these limitations, provided their coproducts are not explicitly nested. By declaring the type constructor `(:+:)` to be right-associative, types such as `f :+: g :+: h` are parsed in a suitable fashion.
>
> (Swierstra 2008)

The expression problem is a such a classical benchmark of expressivity because it showcases the omnipresent use case of the programmer who wants to make his interface evolve in both the data it treats and the operations it wants to perform on it.

As we have seen with the ability to define an apple-picking order *a posteriori* in § 1.1.8 on p. 29, a type class system already help the programmer lacking perfect foresight. With it, he can use the methods of a given class hierarchy (such as `pick`) on objects belonging to another. The situation in which this is not possible — as with vanilla object hierarchies — is often called *the tyranny of the dominant decomposition* (Lämmel and Ostermann 2006). But while this was simply interface conformance, what overlapping instances would provide us is the ability to define an interface (here, our coproducts) which admits new data variants, without having to conform to a particular, pre-existing style. Thus, **a type class system that relaxes this limitation while maintaining complete determinism seems to bring a valuable expressivity bonus to the user.**

Swierstra's exact solution to the expression problem remarks that given a recursive (constructor) function `(f: A → B)`, one can define a second order function called functional `F : (A → B) → A → B` such that `F` is itself non-recursive and `f = F f`. This allows Swierstra to separate concerns in defining his expression type, but, crucially, uses an unbounded recursion operator to thread instance resolution through expressions of arbitrary length — hence the adaptation if this solution in COQ requires manageable, but particularly heavy transformations that fall beyond the scope of this document.[78] But this is an orthogonal concern: however the recursion mechanism is implemented, it is clear that the crux of the expressivity problem of the "à la carte" approach is in the type class instance selection. And this aspect of the solution has indeed garnered enough interest to prompt a call for that exact extension to GHC's overlapping instance selection mechanism — aiming, ultimately, at an inclusion of that advanced feature in the main language. Morris and Jones (2010) indeed aims at defining *instance chains*, a way to define overlapping instances in a precise, continuation-passing style that covers the possible alternatives in a deterministic fashion, using an "`if-then-else`" syntax.

[78] The curious reader will refer to (Bove et al. 2011, §5), which surveys the state of the art of solving that particular issue.

On paper, COQ also prohibits overlapping instances, and will always select the first declared `Canonical` instance for a given (value, projection) pair.[79] However, thanks to key properties of unification mentioned in § 1.2.4 on p. 42 it turns out that those instance chains are already accessible with `Canonical Structures`. Let us see why.

[79] Since the inclusion of a patch by the author in version 8.3, COQ signals it will not use redundant projections for `Structure` inference, should the user declare some.
This technique originates in work by Gonthier and Cohen on implementing decision procedures by reflection in COQ (Cohen 2010, Annexe C). The key insight was ported to a new flavor based on default instances in the reflection of expressions of direct sums of matrices in (Gonthier 2011, §4.3), then developed as a way to systematize lemma application and provide deterministic automation facilities in (Gonthier et al. 2011).

Instance Chains The objective of Morris and Jones (2010) is to upgrade the overlapping instance mechanism to allow the following form of syntax for instance chains in HASKELL:

```
instance f :<: f
else f :<: (g :+ : h) if f :<: g
else f :<: (g :+ : h) if f :<: h
else f :<: g fails
```

We want to emulate the same kind of behavior in COQ. We start with reproducing the coproduct type, so as to have a precise idea of the actors in play (Fig. 1.43).

```
CoInductive Val (T:Type) : Type :=
  Build_Val (n : nat).

Record Add (T : Type) := {
  e1 : T;
  e2 : T
}.

CoInductive Coprd (f g : Type → Type) (T:Type):=
| Inl: ∀(e0:f T), Coprd f g T
| Inr : ∀(e1: g T), Coprd f g T.

Check (@Inl Val Add nat (Build_Val _ 1)).

Structure subtype (sup: Type → Type)
  (supP: phantom (Type → Type) sup) := {
  sub :> Type → Type;
  inj : ∀(T:Type), sub T → sup T
}.

Notation "{ 'subtype' fplusg }" :=
  (subtype (Phantom _ fplusg))
  (at level 0, format "{ 'subtype'  fplusg }")
  : form_scope.

Definition sup (f:Type → Type)
  (k:{subtype f}) : Type → Type := f.

Definition subtype_pack
  (sub : Type → Type) (sup : Type → Type)
  (i: ∀T : Type, sub T → sup T) :=
  Build_subtype (Phantom _ sup) i.
```

Figure 1.43: The definition of a coproduct with a first attempt at a subtype definition.

As in the previous paragraph on adjunctions (§ 1.4.3), we use a phantom type to expose a *required type* of the subtype structure : the right member of the subtyping relation. It allows us to require explicitly terms of a specific coproduct type, using the `{subtype (Coprd f g)}` notation. The *object* of the subtype structure, or the *key* — i.e. the first projection of the structure, which will trigger inference in lemmas and other constructs — is the (smaller) subtype, an expression instance we want to inject in (or see as a member of) a larger coproduct type.

We emulate the two most basic instances declared by Swierstra (2008), including a `Val` decorated with a phantom argument.

The key insight towards making instance chains work in COQ was hinted at in note 48 on p. 44. Let us go over what we said in § 1.2.5 on p. 43: the `Canonical Structure` mechanism, triggered at type inference, searches for a record value fitting a given head constant and a projection. It fires when it encounters an equation that looks like :

$$\texttt{sub}\,?f \triangleq \texttt{Val}$$

Then, it looks in the `Canonical Projections` table, and does head constant matching with the (*projection*, *head value*) pair `(sub, Val)`, hoping to encounter a suitable record declaration. A crucial behavior is that **if and**

```
Definition idconstr (f:Type → Type)
     :=
  fun T (x: f T) => x.
Canonical Structure reflex_subtype
     (f:Type → Type) :=
  subtype_pack (@idconstr f).
Canonical Structure left_subtype (f
     g:Type → Type) :=
  subtype_pack (@Inl f g).
Definition in_subtype_right (g h:Type
     → Type)
  (sfg: {subtype g}) :=
  fun T => comp (@Inr h g T) (@inj _
     _ sfg T).
Canonical Structure right_subtype (g
     h: Type → Type)
  (sfg: {subtype g}):=
  subtype_pack (@in_subtype_right g h
     sfg).
```

Figure 1.44: An erroneous definition of coproduct instances.

only if the head-constant matching fails, it δ-reduces the right member of the equation, and tries again. This single step of δ-reduction is due to the "δ-delaying politic" we have investigated in § 1.2.4 on p. 42.

There can unfortunately be only one record instance (the "value" in the sense of the key-value association of a Map) per (projection, head constant) pair (the "key"). We would like COQ to try several. But the declaration of some instances may mask others. In particular, let us see what happens when we try to define the instances of Swierstra (2008) naively. We obtain what figures in Fig. 1.44 on the preceding page. When we try to declare the `left_subtype` instance, COQ answers with:

```
Warning: Ignoring canonical projection to _ by sub in
  left_subtype:
redundant with reflex_subtype
```

Indeed, `reflex_subtype` is a peculiar instance, meaning an instance for which the inference-triggering value (here the first projection) is a variable. This has two consequences, the first of which shows it is treated in a special fashion, and another for which `reflex_subtype` is just as any other instance:

(i) the value for `sub` of `reflex_subtype` has no head constant, but it is still considered as a valid instance : this is the exception to the rule for inferrable record projection values of always having a head constant, called *default instances*. Those records where the inference-triggering member is a variable guarantee unification with any value always succeed — but they shadow any further value.

(ii) the `sub` projection value for `reflex_subtype` will always make the `sub` projection values of `subtype Canonical Structures` declared *after it* redundant.

Even if our default instance had been declared last, we would have faced the reverse overlap problem, and would have gotten the message above, but with `left_subtype` and `reflex_subtype` inverted.[80]

[80] In our case, what is even worse is that all instances are default instances: though the returned types are distinct, the `sub` projection of all three instances returns a variable.

The central idea to addressing that problem is to make sure the reduction on the right — the one which occurs at the failure of unification of a type class instance — triggers a new, distinct `Canonical Structure` search problem. For that, we use δ-reduction. Our goal is then to craft δ-redexes inside the canonical instances we try to define. Unfolding those redexes, we provide a sort of continuation-passing style semantics to switch between instance alternatives.

To achieve that, we use an ad-hoc boxing structure for constructors:

```
Structure tagged_constr := Tag {untag :> (Type → Type)}.
```

This structure is a pure boxing structure. It begs for a default instance taking a `(h : Type → Type)` as a variable argument — an instance we can bestow on any term of type `Type → Type`. Now, what happens if this instance contains layers of δ--redexes instead ?

```
Definition right_sub (h:Type→Type) := Tag h.
Definition left_sub h := right_coprod h.
Canonical Structure refl_sub h := left_coprod h.
```

We then see the means of implementing our continuation-passing semantics:

```
Structure subtype (sup: Type → Type) (supP:
    phantom (Type → Type) sup) := {
  sub :> tagged_constr;
  inj : ∀T, (untag sub) T → sup T
}.
Definition subtype_pack
  (sub : tagged_constr) (sup : Type → Type)
  (i: ∀T : Type, (untag sub) T → sup T) :=
  Build_subtype (Phantom _ sup) i.

Definition right_sub (h:Type→Type) := Tag h.
Definition left_sub h := right_sub h.
Canonical Structure refl_sub h := left_sub h.

Definition idconstr (f:Type → Type) := fun T
    (x: f T) => x.

Canonical Structure reflex_subtype (f:Type →
    Type) :=
  @subtype_pack (refl_sub f) _ (@idconstr f).
```

```
Definition in_subtype_left (f h: Type → Type)
  (sfg : {subtype f}) :=
  fun T => comp (@Inl f h T) (@inj _ _ sfg T).

Canonical Structure left_subtype (g h: Type →
    Type)
  (sfg: {subtype g}) :=
  @subtype_pack (left_sub (untag sfg)) _
    (@in_subtype_left g h sfg).

Definition in_subtype_right (g h:Type → Type)
  (sfg: {subtype g}) :=
  fun T => comp (@Inr h g T) (@inj _ _ sfg T).

Canonical Structure right_subtype (g h: Type →
    Type)
  (sfg: {subtype g}):=
  @subtype_pack (right_sub (untag sfg)) _
    (@in_subtype_right g h sfg).
```

Figure 1.45: The definition of a subtype type, with instances directed by δ-contraction.

- providing we can infer a `tagged_constr` structure on our object,
- and provided `subtype` can take a `tagged_constr` as a first projection
- we can declare instances of `subtype` that are tried on the most δ--contracted redex above, then on its δ-expansion, etc ...

This is what we implement in Fig. 1.45. We can now see that we have three (projection, head constant) pairs registered as keys in the `Canonical Projections` table:

```
right_sub ← sub ( right_subtp )
left_sub ← sub ( left_subtp )
refl_sub ← sub ( reflex_subtp )
```

In sum, δ-aliasing provides us with a way to script the `Canonical Structure` inference process, guided by failure.

```
Definition clone (f: tagged_constr) (g: Type →
    Type) (cT:{subtype g}) :=
  fun c & phant_id (@sub g _ cT) f & phant_id
    (@inj g _ cT) c =>
  @subtype_pack f g c.

Notation "[ 'subtype' g 'of' f ]" := (@clone
    (refl_sub f) g _ _ id id)
  (at level 0, format "[ 'subtype' g 'of'  f ]")
    : form_scope.
```

```
Check ([subtype Val of Val]).
Check ([subtype Add of Add]).

Check ([subtype (Coprd Add Add) of Add]).
Check ([subtype (Coprd Add Val) of Val]).
Check ([subtype (Coprd Add (Coprd Val Add)) of
    Val]).
```

Figure 1.46: Testing the `subtype` structure.

All that is left is to test that our `Structure` behaves as intended, in Fig. 1.46. For that we define a constructor notation (that we will explain in § 2.3.2 on p. 108). This emulates the presence of a structure projection in a lemma, for instance. On the last line, we see we finally have a subtype structure for `(f :+: g) :+: h`

1.4.5 *Classes and other type class mechanisms*

The type class mechanism[81] provided by `Canonical Structure`s, as well as the one of `Class`es cover from the start some of the extensions traditionally implemented on top of the original type class implementation (Peyton-Jones 2003; Wadler and Blott 1989).

[81] We invite the reader unfamiliar with HASKELL's type class system beyond the Haskell 98 standard to consult the helpful survey

S. Peyton-Jones, M. P. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, 1997. URL http://research.microsoft.com/en-us/um/people/simonpj/Papers/type-class-design-space/

As we have mentioned in the previous subsection (§ 1.4.3 on p. 70), we support multiple parameters, though only `Class`es have the multiple dispatch necessary to ensure an instantiation that uses either key. Meanwhile, the dependently typed calculus offered by Coq ensures we can easily emulate the extensions of the HASKELL type class mechanism that want to bring the system closer to functions between type instances, simply using dependent records : we can refer to previous members of the record to emulate functional dependencies (Jones 2000) and add type parameters to emulate associated types (Chakravarty et al. 2005).

The registration of class instances as a hint declaration over a named first-class record in either paradigm (an example is featured in Fig. 1.19 on p. 52) ensures we support features equivalent to the named instances (Kahl and Scheffczyk 2001) of HASKELL. The higher-kinded polymorphism of COQ, along with the possibility to define `Structure`s or `Class`es dependent on a parameter, lets us define constructor classes natively (Jones 1993).

Now, as previously mentioned in § 1.3.1 on p. 51, COQ has another type-class implementation (Sozeau and Oury 2008). Let us provide some brief remarks as to how they compare to the `Canonical Structure` mechanism. As we have been emphasizing in the whole of this chapter, a type class system is at its heart two mechanisms built on top a generic programming construct:

- a constraint propagation mechanism,
- and an instance selection mechanism.

For both `Class`es and `Canonical Structure`s, this generic programming construct is the same: *dependent records*. We have shown extensively that for `Canonical Structure`s, the instance selection mechanism is an instrumentation of the type inference procedure dealing with record projections (§ 1.2.5 on p. 43), and the constraint propagation mechanism is either :

- telescopic-style record nesting (§ 1.1.10 on p. 32), facilitated by coercions (§ 1.3.4 on p. 56), for small examples that don't need to be scalable,
- or the more efficient (but more verbose) packed classes (§ 1.4.1 on p. 60).

For `Class`es, the constraint propagation mechanism is Pebble-style sharing (§ 1.1.9 on p. 30),[82] and the instance propagation mechanism is the use of the tactic base `eauto` during implicit argument resolution (Sozeau and Oury 2008, §6.4), aided with interactive proof obligation discharge. Note that since `eauto`'s matching is also based on higher-order unification, both instantiation mechanisms share some of the core traits of the distinct unification implementations in the COQ.

As we have shown in § 1.4.2 on p. 67 and 1.4.3 on p. 70, thanks in part to the packed classes idiom, **there is no real issue of *expressivity* on the front of constraint propagation. However**, as mentioned in § 1.4.1 on p. 60, **there is a real issue of *efficiency***: because `Class`es rely more significantly on record parametrization, they can be expected to lead to large terms which do not play well with current COQ procedures and tactics.[83] The instantiation mechanism of `Class`es, on the other hand, bears two important remarks:

- since `eauto` is a tactic that will pick possible instances based on the current context and a hint database, **instance selection is inherently *nondeter-***

[82] See also "Superclasses as Parameters, Substructures as Instances" (Sozeau and Oury 2008, §4).

[83] The reliance on record parametricity has been found to be a considerable performance burden in other type theory-based provers: the compilation of the standard library of Agda 2 (Norell 2007) was recently shrunk by 30% in both time and space once the internal representation of record projections was made to ignore its parametric arguments (Norell 2011): the local type inference approach suggests that projection is easy to tidy, even in a type synthesis setting. We strongly hope for further investigation of this technique in COQ.

ministic **and** ***backtracking***. The second characteristic means that **multiple dispatch and overlapping instances are available natively**: the user can, taking the system as-is, ambiguously associate several categories to a type which would represent their objects, a characteristic used to great benefit by Spitters and van der Weegen (2011). However, controlling which exact instance is applied when the context is not enough to make the type unification of the inappropriate cases fail is more problematic. In particular, since the *instance chain* technique mentioned in 1.4.4 on p. 74 is based on the late application of δ-reduction in COQ's unification algorithm, it can be (and was) adapted to `Class`es (Spiwack 2011). However, as further developments of that technique have shown (Gonthier et al. 2011, §7), it becomes very difficult to control the behavior of unification in advanced cases, restricting this technique to anecdotal examples in the context of `Class`es.

- since `eauto` deals in a backtracking fashion with such an unlimited search space, **instance search can become very slow** in the context of multiple, functional, ambiguous instances.

An additional (and final) difference is that `Class`es resolution is primed at the implicit argument resolution phase, and concluded after type inference. Since it has no ability to dialogue with inference, it makes the instantiation of structures such as the tuples presented in § 1.3.4 on p. 56 much harder, especially in cases where COQ would have to invert conversion to find a suitable instance (e.g. finding an instance for `tuple (4 + 4)`). As a more anecdotal use, the user can define `Canonical Structure`s based on a sort in COQ (an elegant way to provide ad-hoc type reifications[84]), which requires an awkward translation using `Class`es since implicit argument resolution occurs before type inference, and as such has no access to, *e.g.*, coercions. While overcoming that last hurdle may seem a complex endeavor with as yet no clear payoff, the eager resolution of value classes has on the contrary been of considerable use in theory development (see the vector types of Gonthier 2011), and we hope this limitation will could be addressed in further developments of `Class`es (potentially by marking arguments for which eager unification by a provided value should be triggered on the spot).

[84] The curious reader will consult `AdvancedCanonicalStructure.v` and `AdvancedTypeClasses.v` in the `test-suite/success/` directory of COQ's source distribution.

Implementation

2

THE FUNDAMENTALS OF THE STRUCTURES developed in SSReflect come up in several publications in the literature. The sixth section of (Gonthier and Mahboubi 2010) presents — after a tutorial on small scale reflection and some indications on the use of `Canonical Structure`s — the way to use *finite types* in SSReflect. Those indications update the first half of (Gonthier et al. 2007), which dealt with preliminary versions of more of the finite structures of the SSReflect library. The next most recent description of those structures appears in (Ould Biha 2010), which covers, along the path towards a treatment of representation theory, a description of indexed operations already touched upon in (Bertot et al. 2008). A more generally-scoped update of the abstract algebra developed while defining the basic concepts needed for character theory figures in (Garillot et al. 2009). The updates to the SSReflect manual for the 1.3 release also feature a map of interfaces defined in the libraries (Gonthier et al. 2008, §10).

Along this ante-chronological walktrough of SSReflect-related literature, however, we need to mention that the second half of (Gonthier et al. 2007) has known other significant updates. Those updates go beyond the restructuration brought by the *packed classes* discipline we touched upon in the previous chapter[1] and yet, even the packed classes improvement was not shown brought to bear on simple group structures. In this chapter, we intend to fill these gaps. Namely,

- We intend to give some insight into trademark design choices for the SSReflect definitions and library of `Structure`s. For this, we will pick examples among the modern forms of some of the finite structures described in (Gonthier and Mahboubi 2010; Gonthier et al. 2007) but will not aim at exhaustivity.
- To give a better sense of the development process, we expand to applications I developed in the SSReflect library: cyclic groups.
- This will allow us to introduce how the collision between a prior notion of morphism, and the newly developed automorphisms made us improve the treatment of `Structure`s with a parameter. We delve on the consequences for the formalization of partial functions in the wild.
- Incidentally, this bedrock of use cases will allow us to show how the user can, in COQ, direct the `Canonical Structure` inference process to allow for richer and more succint context-dependent functions and proofs.

[1] Garillot et al. (2009) chose to feature a set of examples distinct from the one used by Gonthier et al. (2007) to present the upheaval.

2.1 Groups, sets and structures

THE SSREFLECT LIBRARY IS A LARGE FORMALIZATION (more than 100 000 lines of source code at the time of this writing), and therefore relies on a few, simple, consistent design choices in the approach of basic mathematical objects such as sets, functions and finiteness. These choices occur at the heart of a formalization: we are talking here of the computational definitions, in the programmatic sense, that we will consider as the representants of the simplest — and therefore most pervasive — definitions of our system.

Those choices matter since we work on the assumption that proofs have, like programs, their primitive types, user-defined but yet akin to the naturals supported by any programming language: below a certain level of refinement, it is the *computational* behavior of a term that gives a convincing representation of a mathematical object whose mathematical textbook definition is itself, after close examination, computational. A simple example: denumerable sets are defined by an effective indexation to the naturals (Bourbaki 2006, §7.9), a notion that involves an embedding, which has to be accessed at some point in a formalization. Not only is it important for us that those first definitions be practical, but we also wish those simple objects to give an intuitively convincing image of the foundations of our hierarchy: the relation to the original mathematical object should be trusted as being *"so simple that there are obviously no deficiencies"*.[2]

[2] C. A. R. Hoare. The emperor's old clothes. *Communications of the ACM*, 24 (2):75–83, Feb. 1981. ISSN 00010782. doi:10.1145/358549.358561

Finally, those choices are naturally equally important from a software engineering point of view: programmatic tools such as modules or type class mechanisms can help the user define translations between representations, but using them constantly across a large development means that an unreasonable fraction of the proving and typechecking time could be wasted on the computational side effects of applying that translation.

2.1.1 Boolean reflection

Proof by reflection in COQ[3] is a method for studying properties (i.e., types) using a term of the language. The study of that *reflect* gives a simplified representation of the property, since the reflect captures only the parts of the property's objects necessary for building a proof. In particular, when such a property is *decidable*, a *decision procedure* can be given such a reflect as an argument. It is then enough to prove that the result of that procedure can systematically be used to build a proof of the property — or of its logical negation.

[3] We invite the reader longing for a refresher on this technique to consult the extensive primers found in the literature (Bertot et al. 2008; Chlipala 2009, resp. §16, §13).

More specifically, boolean terms provide perfect reflects for decidable properties, for which the excluded middle holds. Moreover, helper lemmas applicable on boolean formulas, especially rewriting lemmas, are easy to define and use. Hence, to use the boolean reflection method in a systematic fashion for decidable properties, we use the following inductive:

```
Inductive reflect (P : Prop) : bool → Set :=
  | ReflectT : P → reflect P true
  | ReflectF : ~ P → reflect P false.
```

Though this predicate caters to boolean reflection, i.e. towards dealing with properties for which booleans provides good reflects, it is easy to imagine how to adapt it to the case where another type of terms describes the set of reflects we want to study.

Having an instance of this inductive provides a guarantee of the equivalence of a term `b : bool` and a proof `p : P`. Destructing this inductive therefore provides the user with a fast way to commute between proposition `P` and `b = true`[4], in a systematic and fast fashion. In SSReflect, we call this mechanism a view, and it has led to the common practice of *small scale* reflection throughout the SSReflect libraries, described in Fig. 2.1.

[4] The SSReflect library uses the systematic syntactic sugar of displaying this equality as if it was the boolean itself using a coercion: `Coercion is_true b := b = true.`

Let us consider a highly complex boolean combination of decidable properties. Simplifying the easily eliminable subterms of this formula using the usual Coq tactics leads to an untractable proliferation of cases, that only a complex database of `Ltac` tactics can handle. The SSReflect user will rather transform this formula into a complex *boolean* formula, for which the elimination of cases consists in rewriting with a set of well-chosen lemmas. This does not, however, mean that the user has to do full-scale reflection and reduce the whole formula to `true` or `false` in one go. The user will often decide to treat relevant terms, after simplification, through the regular Coq machinery of tactics. Using a view again, he can finish the proof of his property, expressed now as a *simplified* logical formula. This is different from the general usage of proof by reflection, where one expects a custom decision procedure to be the end-all of the processing of a reflect towards the proof. The *second* view application in our figure is what makes us call this reflection pattern *small-scale*.

In summary, **boolean reflection with views is a fast tool to simplify complex decidable properties using rewriting lemmas.**

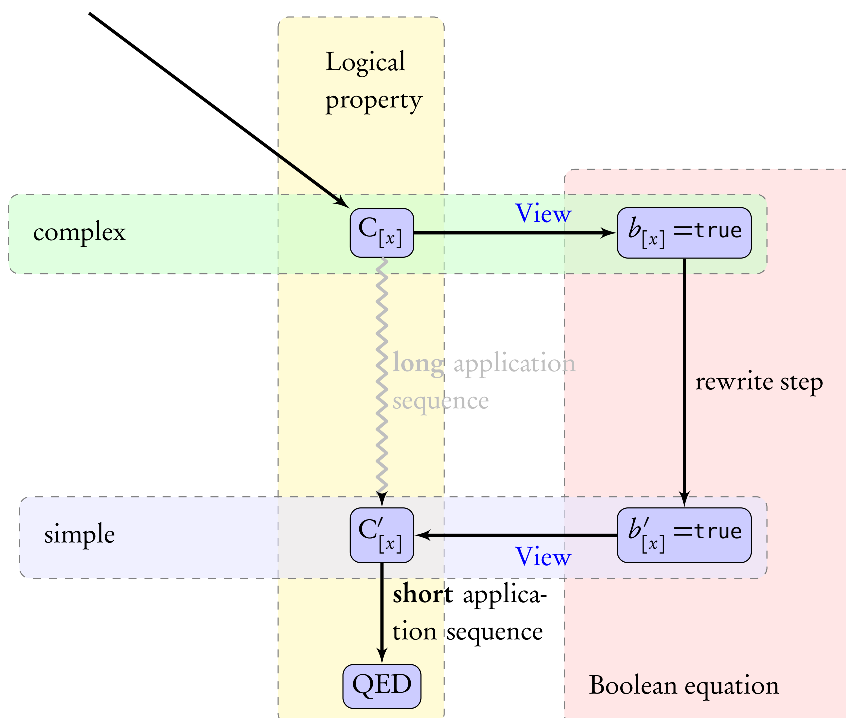


Figure 2.1: SSReflect encapsulates boolean reflection in its compact **view mechanism**

Thus, in the SSReflect library, if the user wants to study a property which is inherently decidable, it is strongly favored to define it directly as a boolean predicate related to the actual desired proposition type with a reflection lemma, rather than as a property. An example follows.

2.1.2 *Types with a decidable equality*

It is well-known that in constructive type theory, and in particular in COQ, we do not enjoy proof-irrelevance for the `Type` hierarchy. For example, a Σ-type `{ x:A | P x }` (those of the standard COQ library, as featured in § 1.1.6 on p. 25) can have two inhabitants with the *same* witness and equipped with *distinct* proofs of the *same* property.

This holds even for equality proofs, for which adding proof *irrelevance* is equivalent to adding Altenkirch & Streicher's K axiom,[5] which is independent from the Calculus of Constructions. But there is an important subcase of proof irrelevance that always holds. Streicher's axiom K always holds on *decidable domains*, *i.e.* equality proofs on sets where equality is decidable are unique. The general proof is in the `Eqdep_dec` module in the COQ standard library.

[5] T. Streicher. *Semantical Investigations into Intensional Type Theory*. Habilitation thesis, Ludwig-Maximilians-Universität München, 1993. URL http://www.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf

This means that, provided we are on a type with decidable equality, we can recover some of the proof irrelevance that is so intuitive when doing classical mathematics, without adding anything to the calculus. However, proving decidable equality for a new, custom type involves threading the preservation of the decidability property along applications of its constructors: a trivial, but tedious task. For this, we let the compositionality of type classes help us. We define a generic structure for types with a decidable

```
Definition pred T := T → bool.
Definition rel T := T → pred T.

Module Equality.
Definition axiom T (e : rel T) := ∀x y, reflect (x = y) (e x y).
Structure mixin_of T := Mixin {op : rel T; _ : axiom op}.
Notation class_of := mixin_of (only parsing).
<...>
Module Exports.
Notation eqType := type.
Notation EqMixin := Mixin.
Notation EqType T m := (@pack T m).
End Exports.

End Equality.
Export Equality.Exports.
```

Figure 2.2: `eqType` : a type equipped with a decidable equality. We note the boolean equality operator of this structure with the infix symbol `==` .

equality[6] in Fig. 2.2. The axiom of this concept shows that the *boolean* equality function defined therein is a *reflect* of the Leibniz equality. It is an instance of the `reflect` predicate seen in the previous paragraph, and as such, allows us to convert quickly (using *view*), between the predicate `(Equality.axiom _ x y) = true` and `x = y`. This is particularly useful since the latter is the only relation with which the user can natively rewrite in COQ: we have promoted an arbitrary equality procedure to a rewritable relation.

[6] We use the packed classes idiom described in § 1.4.1 on p. 58. Therefore **we do not give the full source of the definition of `eqType`, as for the remaining types of this document. We rather limit ourselves to the `mixin` part, and, when appropriate, to the `class` record relevant to those types.** The rest can is unambiguously inferrable from our discipline, and the full source is readable in the SSReflect libraries.

This class also allows us to make use of a generic proof-irrelevance result, akin to that featured in `Eqdep_dec`, but which uses `reflect` to build the

canonical proof of equality one needs to provide in such an irrelevance theorem.

```
Theorem eq_irrelevance :
  ∀(T : eqType) (x y : T) (e1 e2 : x = y), e1 = e2.
```

Given that result, we can look, specifically, at the instance of it for booleans in Fig. 2.3.

Finally, boolean proof irrelevance lets us build a class of *subtypes*, analogous to Coq's Σ-types, but paired with a decidable, boolean predicate instead of a general one.

```
Lemma eqbP : Equality.axiom eqb.
Proof. by do 2 case; constructor. Qed.

Canonical Structure bool_eqMixin :=
  EqMixin eqbP.
Canonical Structure bool_eqType :=
  Eval hnf in EqType bool
     bool_eqMixin.
```

Figure 2.3: The declaration of an `eqType` for booleans

```
Variables (T : Type) (P : pred T).

Structure subType : Type := SubType {
  sub_sort :> Type;
  val : sub_sort → T;
  Sub : ∀x, P x → sub_sort;
  _ : ∀K (_ : ∀x Px, K (@Sub x Px)) u, K u;
  _ : ∀x Px, val (@Sub x Px) = x
}.

Implicit Arguments Sub [s].
Lemma vrefl : ∀x, P x → x = x. Proof. by []. Qed.
```

Figure 2.4: A decidable `subType`, coercible injectively to its parent.

The `subType` definition[7] is in Fig. 2.4. It defines a structure giving a generic name `val` for the projection to the parent type, given defining elements of a decidable subtype that it is *isomorphic to* : a constructor `Sub`, an eliminator, and an injectivity proof for the `val` projection [8]`{ x | P(x)}` for a boolean predicate `P`. `subType` instances are intended to be built most of the time using the notation:

```
Notation "[ 'subType' 'for' v 'by' rec ]" := (SubType _ v _
    rec vrefl)
  (at level 0, format "[ 'subType' 'for' v 'by' rec ]") :
    form_scope.
```

Herein, `v` plays the role of a record field projection, with `rec` the induction principle generated by Coq during `Inductive` or `Record` declarations. Note `vrefl` (Fig. 2.4) provides the canonical reflexivity witness (through the trivial tactic), ensuring the projection really is the inverse of the first argument of the constructor found in the elimination scheme. The instances of this type enjoy proof-irrelevance on the second parameter (of type `P = true`), and the *injectivity of their first projections*. Thanks to that last feature, `subType` instances inherit exactly the `eqType` comparison of their parents after forgetting the proof witness, which reduces to naught the difficulty inherent in dealing with general dependent Σ-types.

[7] (Garillot et al. 2009; Gonthier and Mahboubi 2010, resp. §3.1,§6.2) treat the API and usage of that construction extensively.

[8] Remember the official Coq existential construction of § 1.1.6 on p. 25 is a specialized instance of a more general construction (dependent pairing through inductive boxing) that we want to support in all its generality. We then require a projection and an elimination scheme, *as defined using the generic constructor inverse to that projection*. See the generic `subType` instance for `sig` in the library for more details.

2.1.3 Sets as collections

Another design paradigm in the SSReflect libraries is to approach the notion of sets as a selection of elements over a type. This choice is a necessary one: while mathematicians rarely make their choice of foundations explicit, we do not have the option to be that lenient. Meanwhile, this is a non-obvious choice, because type theory does not in itself provide a single equivalent to the notion of set found at the heart of Zermelo-Fraenkel-based mathematics.

One alternative is to consider that in type theory, types themselves are a somewhat appropriate notion to formalize the notion of set. The definition of a new set therefore corresponds to the definition of a new type. For instance, this new type can be a co-product of two types, to model the union of heterogeneous sets, or a Σ-type, to model a subset.

We have seen in the previous paragraph that it can be somewhat clumsy to deal with subsets through the inductive construction of Σ-types suggested in the COQ library. And indeed, in the modern calculus of COQ, equipped with little proof irrelevance, this is an option that is rarely adopted. Ubiquitous Σ-types would indeed make transferring the properties of a set to one of its subsets a complex boxing operation, not to mention the heaviness involved in dealing with highly dependent types.

Still close to that flavor of taking types as the primary construct stands the approach of Bishop's set theory.[9] Diverging voluntarily from the usual notions of set theory, it uses types equipped with a relation defining the equality of their elements to describe what the mathematician thinks of as sets, calling them *setoids*. The concept was translated to type theory and COQ,[10] and it figures pervasively in the real analysis developed in the C-CoRN library (Cruz-Filipe et al. 2004). It was also used to formalize some abstract algebra (Yu et al. 2003) in a more axiom-free context than C-CoRN.

[9] E. Bishop. *Foundations of constructive analysis*. McGraw-Hill Book Co., New York, 1967

[10] (Barthe et al. 2003) includes a survey of approaches to adapting this notion to type theory. The COQ implementation of the machinery behind the use of setoids was recently reimplemented. Sozeau (2009) includes comparisons with the previous iterations along with a detailed treatment.

However, the SSReflect library went with an explicit representation of sets, meaning that the sets are instances of a *container type* (in the informal sense) defined itself parametrically on a *type of elements* — that is, its carrier. This is because the proof of the Feit-Thompson theorem sits at a particular location in the realm of formalized mathematics.[11] Indeed, the choice between representing sets as explicit enumerations of elements, or as types with a relation (setoids) really hinges on whether sets themselves — rather than their elements — are the objects of the mathematical discourse. Talking about Bishop sets in proofs — hence if using setoids, *types* themselves — can become clumsy, because it once again brings us back to dealing with dependent pairings such as the ones involved in Σ-types. Finite group theory, meanwhile, quickly turns the spotlight on groups and subgroups, rather than on their elements. This trend is even apparent in the earliest stages of the proof: the isomorphism theorems feature only set-lifted constructions and operations, for instance.[12] Forcing sets to stay concrete thus ensured those common objects of our proofs stayed at the lowest type universe, and were easy to manipulate.

[11] External considerations should also be acknowledged here: the choices for set-theoretic development in the SSReflect library happened in 2006, at which time `setoid_rewrite` had a previous (Sacerdoti Coen 2006), less complete (Sozeau 2009, §4.3) implementation.

[12] One of them appears in § 1.3.3 on p. 54.

Finally, the Mathematical Components team also decided against alternatives giving the primacy to types since it places itself in a finite setting. In this framework, the inhabitants of types can be extensively enumerated, and we wanted to take as much advantage of that property as possible. We will present the benefits we recovered from this in the next section.

Independently of the Feit-Thompson formalization in COQ, the "sets as collections of elements" paradigm has known some popularity, not only in the formalization of mathematics (Arthan 2006a; Bailey 1998a; Gunter 1989), but also as the hallmark of some of the more traditional proofs of program verification or of meta-theoretic properties of programming languages. In fact, it crops up as soon as developments manage their own sets "by hand" — examples include all solutions surveyed by Aydemir et al. (2008)

— thus making the representation of sets with structure a challenge universal enough to warrant careful study.

Hence, before looking at the definition of sets, as defined by an instance of some generic container type, we look at how to define the *parameter* of that container, *i.e.* the carrier type.

2.1.4 Finite Types

```
Module Finite.

Section RawMixin.
Variable T : eqType.
Definition axiom (e : seq T) :=
  ∀x, count (@pred1 T x) e = 1.

Record mixin_of := Mixin {
  mixin_base : Countable.mixin_of T;
  mixin_enum : seq T;
  _ : axiom mixin_enum
}.
End RawMixin.

Section ClassDef.

Record class_of T := Class {
  base : Choice.class_of T;
  mixin : mixin_of (Equality.Pack base T)
}.

End ClassDef.
<...>
Module Import Exports.
Coercion base : class_of ↦ Choice.class_of.
Coercion mixin : class_of ↦ mixin_of.
Notation finType := type.
Notation FinType T m := (@pack T _ m _ _ id _
     id).
End Exports.

End Finite.
Export Finite.Exports.
```

Figure 2.5: Finite types, with explicit enumeration

The definition of our notion of finite type appears in Fig. 2.5. As usual, we do not give the actual `type`, or `pack` occurring in the notations exported by the module `Finite`, since they can be deduced from our packaging discipline.[13] The finite type mixin requests a sequence of elements `e`:

- `(pred1 T x) : T → bool`, where `T` is an `eqType`, is the predicate that returns `true` if and only if its argument is equal to `x`
- `(count a s) : nat` counts the number of elements of `(s : seq T)` that verify `(a : pred T)`[14]

Therefore, the `axiom` requested in our finite type structure is that `e` exactly enumerates the elements of our finite type.

Since we have that enumeration, we can give a cardinal to our finite type. The definition presents no particular difficulty, so that we just present the notation we will use henceforth (Fig. 2.6). Having this cardinal means that functions defined on a finite type have a bounded domain: they can therefore be tabulated. It is easy to write a type for such tabulations: it amounts to a sequence of elements of the range type, of length the cardinal of the domain, and which definition features on Fig. 2.7 on the following page.[15] To actually apply the implied function on an element `(x : fT)`, with `fT` a finite type, it is enough to find the rank of `x` in the enumeration of `fT`, and return the element of the tabulation sequence having the same rank. This operation is made by the function `FunFinFun.fun_of_fin` in Fig.2.7. To close the gap with the usual Coq functions (and use them in functional position in Coq expressions), we simply make this function a coercion to `Funclass`. Incidentally, this formalization allows us to prove extensionality on those finite functions, another feature that brings us closer to the classical framework, without adding any axions. We can now define a set of elements as a selection of elements of the carrier type. We do this by way of the explicit tabulation of an indicator function, on the

[13] We ask for the patience of the reader as to the exact definition of `pack`. It will find its explanation in § 2.3.2 on p. 111. As previously noted, `seq` is a richer reimplementation of the standard library on lists.

[14] The definition of `pred` appears in Fig. 2.2 on p. 86.

```
Notation "#| A |" := (card (mem A))
  (at level 0, A at level 99,
   format "#| A |") : nat_scope.
```

Figure 2.6: Cardinal notation for finite types

[15] We reuse the `tuple` type whose definition is in Fig. 1.23 on p. 57, and discussed in 1.3.4. More details about this construction are in (Gonthier and Mahboubi 2010, §6.3)

We again implore the patience of the reader as to the `phant`/`Phant` use in `finfun_of`, it will be touched upon in 2.3.2. For now, it is sufficient to consider that it is a construction that permits the extraction the first and second arguments `aT`, `rT` of the constructor `Finfun`, from an arrow type `aT → rT`:hence{`ffun (aT → rT)`} means exactly `Finfun aT rT`.

```
Variables (aT : finType) (rT : Type).

Inductive finfun_type := Finfun of #|aT|.-tuple rT.

Definition finfun_of of phant (aT → rT) := finfun_type.
Identity Coercion type_of_finfun : finfun_of ↦ finfun_type.
Notation "{ 'ffun' fT }" := (finfun_of (Phant fT))
  (at level 0, format "{ 'ffun'  '[hv' fT ']' }") : type_scope.

Notation fun_of_fin := FunFinfun.fun_of_fin.
Coercion fun_of_fin : finfun_type ↦ Funclass.
```

Figure 2.7: SSReflect's finite function type.

whole finite type. This indicator function returns a boolean for each element of the carrier, and therefore has a type `{ffun (T → bool)}`, as defined in Fig. 2.8. The user of the SSReflect library will verify that this

```
Variable T : finType.

Inductive set_type := FinSet of {ffun pred T}.

Definition set_of of phant T := set_type.
Identity Coercion type_of_set_of : set_of ↦ set_type.
Notation "{ 'set' T }" := (set_of (Phant T))
  (at level 0, format "{ 'set'  T }") : type_scope.
```

Figure 2.8: SSReflect's finite set definition.

definition of a set as an indicator function gives us particularly smooth definitions for the usual set-theoretic operations. Indeed, we can define a function `finset : ∀T : finType, pred T → {set T}` that returns the set associated to a predicate on a finite type, simply by assuming this predicate is the membership of that set. Moreover, unboxing the indicator function included (via `ffun`) in a set allows us to see this set as a (membership) predicate. In fact, the SSReflect library allows this through coercions, using a special class designed to write predicates in a "collective" form (Gonthier and Mahboubi 2010, §5.3), so that we can write `(x ∈ A)`, for the boolean saying whether `(x : T)` belongs to `(A : {set T})`.

Given this, the definition for the union comes easily (the necessary notations are in Fig. 2.9), and looks precisely like what one might find on a blackboard:

```
Variable T : finType.
Implicit Types A B : {set T}.
Implicit Type x : T.

Definition setU A B := [set x | (x ∈ A) || (x ∈ B)].
```

```
Notation "[ 'set' x : T | P ]" :=
  (finset (fun x : T => P))
  (at level 0, x at level 69,
    only parsing) : set_scope.
Notation "[ 'set' x | P ]" :=
  [set x : _ | P]
  (at level 0, x at level 69,
    format "[ 'set'  x  |  P ]") :
     set_scope.
```

Figure 2.9: Set-from-predicate notations.

Sets are therefore explicit selections of elements of a type, given through an indicator function, which is often itself the composition of other set-defining predicates. Let us now move on to the run-of-the-mill objects of an algebraic hierarchy: set-theoretic constructs with abstract operations and properties.

2.1.5 Finite Groups

When the notion of group comes up outside of mathematically-inclined circles, it is as the mathematical tool of choice to study the symmetries of concrete objects. This is an implicit reference to the representation theory of groups, which consists, instead of looking at the multiplication structure

within the group, in looking at how a group can *act* on another object. In particular, the group of nonsingular $n \times n$ matrices with real coefficients acts by matrix multiplication on n-dimensional vectors. By mapping the law of a group to the general linear group on n-dimensional vector space V, the group actions provide invertible linear transformations on V.

For the reader interested in how the SSReflect libraries tackle modular representation theory, we suggest better references than this document.[16] But it remains clear that the key requirement for the group law to capture a "symmetry" is to feature transformations which leave invariant the object acted upon, or, equivalently, to feature *invertible* transformations. Thus, the way that we —along with classes and textbooks on the subject — introduce the notion of group is the following:

[16] S. Ould Biha. *Composants mathématiques pour la théorie des groupes*. Thèse de doctorat, Université de Nice - Sophia Antipolis, 2010. URL http://tel.archives-ouvertes.fr/tel-00493524/en

Definition 2 (Group – Mac Lane and Birkhoff 1988)**.**

A group G *is a set* G *together with a binary operation* $G \times G \to G$*, written* $(a, b) \mapsto ab$*, such that:*

(1) This operation is associative.

(2) There is an element $u \in G$ *with* $ua = a = au$ *for all* $a \in G$.

(3) For this element u*, there is to each element* $a \in G$ *an element* $a' \in G$ *with* `aa' = u = 'aa`.

In other words, a group is a monoid in which every element is invertible.

The group is, of course, finite when the underlying set is. The *operation* is also called the *law* of the group, and u its *identity element*. Note the explicit overloading of the letter G above.

To figure out how to map this definition to a `Definition` in COQ, we can notice that the "sets as collections" paradigm goes further than dictating how we specify the basic set-theoretic concepts. Indeed we notice that when mathematical properties speak of group-theoretic relations (such as the *subgroup* relation) they usually relate groups which elements have at least a common type. Those groups, without explicit mention, obviously share the *same* notion of group operation and identity element. When this is not the case, and when the studied relations are therefore heterogeneous, the mathematical statement always features an explicit mapping of some sort (recall the isomorphism theorem example of § 1.3.3 on p. 54).

Hence it is helpful in the definition above to separate the *set* G from the companion law and identity element. We thus let the carrier type carry as much of the algebraic structure as we can. The declaration for the finite group type mixins is in Fig. 2.10 on the next page. It is an evolution of the definition of *finite group domain* provided in (Gonthier et al. 2007), with an added, textbook case of *thin-slicing* (§ 1.1.9 on p. 30). Indeed, the group domain type structure now factors through another structure referred to in the mathematical literature as a semigroup with involution or $*$-semigroup: a "pregroup" (noted as `base_type`), *i.e.*, a monoid with an involutive antimorphism.[17]

When the two parts of this formalization are joined, as in `type`, which joins a finiteness condition and group operations through `base_type` and an inverse property, it is equivalent to the traditional characterization of the properties of the finite group type (Gonthier et al. 2007, §3.2). However,

[17] An antimorphism g is an operation defined on monoids that reverses the order of multiplication: $\forall x, y, f(x * y) = f(y) *' f(x)$. The Mathematical Components library customarily adopts a specific, but rather involved notation for morphism properties.

The usual property saying morphism f is compatible with multiplications of its domain and range — that we will detail in § 2.2.3 — says: $\forall x, y, f(x*y) = f(x) *' f(y)$. We note this by stating the operations f has to be compatible with : `{morph f : x y / x * y >-> x *' y}`. When we work in an endomorphic setting where $* = *'$, we elide the second mention: `{morph f : x y / x * y}`. As a consequence, the property of anti-endomorphisms is stated: `{morph f : x y / x * y >-> y * x}`.

```
Record mixin_of (T : Type) : Type := BaseMixin {
  mul : T → T → T;
  one : T;
  inv : T → T;
  _ : associative mul;
  _ : left_id one mul;
  _ : involutive inv;
  _ : {morph inv : x y / mul x y ↦ mul y x}
}.

Structure base_type : Type := PackBase {
  sort : Type;
   _ : mixin_of sort;
   _ : Finite.class_of sort
}.

Definition mixin T :=
  let: PackBase _ m _ := T return mixin_of (sort T) in m.

Definition finClass T :=
  let: PackBase _ _ m := T return Finite.class_of (sort T) in m.

Structure type : Type := Pack {
  base : base_type;
  _ : left_inverse (one (mixin base)) (inv (mixin base)) (mul
    (mixin base))
}.
```

Figure 2.10: The definition of a finite group domain, or `finGroupType`, through a pre-group inherited by *group subsets*.

the `base_type` also provides some properties still valid on group subsets.

With this structure, we can easily reason with subsets of group elements, for example:

- set product can be denoted `A * B`, and the unit set is denoted by `1`.
- left and right cosets are just notation for product with singletons.
- the set monoid uses the same lemmas (`mulgA`, `gsimp`, etc) as the group elements.

As in (Gonthier et al. 2007), we then define groups as sets solely required to contain the *unit* and closed by the application of an *associative operator*, those two last notions being carried by the type of the elements. Along with intermediate definitions controlling coercion classes necessary to state equalities arising from groups in a set context, this is what is represented in Fig. 2.11. This pattern was discovered independently by Arthan (2006a,b), and is so effective that we have since scaled this approach to a much bigger algebraic hierarchy (Garillot et al. 2009).

```
Module GroupSet.
Definition sort (gT :
     baseFinGroupType) :=
  {set gT}.
End GroupSet.

Definition group_set A :=
  (1 ∈ A) && (A * A ⊆ A).

Lemma group_setP A :
  reflect (1 ∈ A ∧
    {in A & A, ∀x y, x * y ∈ A})
  (group_set A).
Proof.
<...>
Qed.

Structure group_type := Group {
  gval :> GroupSet.sort gT;
  _ : group_set gval
}.
```

Figure 2.11: The definition of a finite group `grouptype`: a or set containing the identity and closed by multiplication.

2.2 Cyclic groups

REASONING ON THE CORRECTNESS OF THE RSA CRYPTOSYSTEM IS MUCH EASIER WITH A LITTLE BIT OF GROUP THEORY. This statement is true when thinking of the *mathematics* of the system, as we will see in the next few paragraphs with a view on the history of proofs of the main theorems and lemmas this correctness property is built upon. But we would also like to make it a claim about the impact of abstraction in the formalization of mathematics: this is a perfect example of a case where the "practical" applications of proof assistants — here, reasoning on cryptographic protocols — can be greatly helped with libraries dealing with abstract algebra, including SSReflect. This thus presents one of the first developments made by the author along this thesis.

Let us look at the encryption and decryption primitives of RSA: given a message, M, represented as a number, the cypher text C is given by $C \equiv M^e \pmod{n}$. Likewise the decryption is defined as $M = C^d \pmod{n}$. The public key is the pair (e, n) and the private key is the pair (d, n), with the message M chunked and padded appropriately so that $M \leq n$.

The correctness property verifies that the cypher text allows us to recover the original message, *i.e.*:

$$C^d \equiv M^{ed} \equiv M \pmod{n} \tag{2.1}$$

For that, it suffices to choose e, d so that :

$$ed \equiv 1 \pmod{\Phi(n)} \tag{2.2}$$

where Φ designates **Euler's totient function**.[18] This is an easy process: for *security* reasons,[19] n is chosen to be the product of two randomly chosen large prime numbers p,q. Since for all n_1, n_2 coprime,

$$\Phi(n_1 n_2) = \Phi(n_1)\Phi(n_2) \tag{2.3}$$

the totient of n is the easily computed to be $\Phi(pq) = (p-1)(q-1) = n - (p+q) + 1$. Hence, to pick e, d verifying (2.2) above, all that remains to do is to pick a random e coprime to $\Phi(n)$, such that we can use Euclid's algorithm to compute d, its multiplicative inverse modulo $\Phi(n)$.

And finally, (2.2) leads to (2.1) thanks to Euler's theorem:

Theorem 1 (Euler). *For all n and a coprime to n, $a^{\Phi(n)} \equiv 1 \pmod{n}$*

Hence, if (2.2) is true, $\exists k$ such that $ed = k\Phi(n) + 1$, so that

$$M^{ed} = M^{k\Phi(n)+1} = M^{k\Phi(n)}M \equiv M \pmod{n}$$

is a consequence of Euler's theorem is M and n are coprime. If they are not (and if $M \neq x$, in which case this is trivially true), we can assume, without loss of generality that $p|M$ and that q and M are coprime. In that case $M^{k\Phi(n)}M \equiv M \pmod{q}$ is a consequence of Euler's theorem, and $M^{k\Phi(n)}M \equiv M \pmod{p}$ is trivially true, which allows us to conclude using a Chinese lemma (see below).

We want to show that a little bit of abstract algebra is key to easily understanding and proving (2.3) and Euler's theorem.

[18]

Definition 3 (Euler's totient function). *For all $n \in \mathbb{N}$, $\Phi(n)$ is the number of integers $1 \leq m < n$ such that m and n are* coprime *(i.e. $m \wedge n = 1$).*

For p prime, we notice that $\Phi(p) = p - 1$.

[19] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21 (2):120–126, Feb. 1978. ISSN 00010782. doi:10.1145/359340.359342

2.2.1 Proving Euler's theorem and properties of $\Phi(n)$.

Elementary proofs The original, elementary proof Euler's theorem[20] only depended in the notion of multiplicative inverse:

[20] Featured in Euler's *Elements of Algebra* (1761).

Proof: Theorem 1. Since a is coprime to n it has a multiplicative inverse modulo n. If we multiply by said inverse an element of the class of am_k modulo n — with m_k one of the $m_1,\ldots,m_{\Phi(n)}$ integers less than and coprime to n — we get an element of the nonzero class of m_k. Hence, modulo a permutation, the $\Phi(n)$ distinct congruence classes modulo n of the $(m_k)_{1\le k\le\Phi(n)}$, are the same as those of the $(a\,m_k)_{1\le k\le\Phi(n)}$. This means that

$$\prod_{1\le k\le\Phi(n)} am_k \equiv \prod_{1\le k\le\Phi(n)} m_k \pmod n$$

Cancellation of the invertible $m_1,\ldots,m_k,\ldots,m_{\Phi(n)}$ from both sides then yields the theorem:

$$a^{\Phi(n)} \equiv 1 \pmod n$$

□

Relation (2.3) was also proved by Euler in 1761 (*ibid.*), in a proof which elementary flavor is best rendered by Kronecker.[21] It uses the preliminary result that:

[21] *Vorlesungen über Zahlentheorie*, (1901), p.245

$$\sum_{d|n}\Phi(d)=n \tag{2.4}$$

This equality is elementarily justified by considering subsets of the n-element set $\{1,\ldots,n\}$, containing those elements that share the same greatest common divisor g with n. Those $Q_g=\{k|1\le k\le n, n\wedge k=g\}$ form a partition:

$$\{1,\ldots,n\}=\uplus_{g|n}Q_g$$

For any $k\in Q_g$ there is a k/g coprime with n/g, and conversely, for any m coprime to n/g, $mg\in Q_g$, so that $|Q_g|=\Phi(n/g)$ and that the egality follows.

Proof: (2.3). We prove $\Phi(mn)=\Phi(m)\Phi(n)$ by strong induction on $(de<mn)$ with m,n coprime.

Thanks to (2.4), we have:

$$\begin{aligned} mn =&\Phi(m)\Phi(n)+\Phi(m)\sum_{d|n,d<n}\Phi(d)+\\ &\Phi(n)\sum_{e|m,e<n}\Phi(e)+\sum_{d|n,d<n,e|m,e<m}\Phi(e)\Phi(d)\end{aligned} \tag{2.5}$$

Moreover since m,n are coprime:

$$mn=\sum_{f|mn}\Phi(f)=\sum_{d|n,e|m}\Phi(de)$$

Applying the induction hypothesis to the lesser terms of the sum, this factors to:

$$\begin{aligned} mn =&\Phi(mn)+\Phi(m)\Phi(m)\sum_{d|n,d<n}\Phi(d)+\\ &\Phi(n)\sum_{e|m,e<n}\Phi(e)+\sum_{d|n,d<n,e|m,e<m}\Phi(e)\Phi(d)\end{aligned}$$

And we conclude cancelling with (2.5). □

A more modern approach Both previous proofs, though elementary, do not have the simplicity of the modern, algebraic approach to the theorem and equality (2.3). The first approach towards simplifying those proofs came with noticing the relationship between $\Phi(n)$ and the *Chinese remainder theorem*, stated as an algorithm to solve linear congruence equations .

Theorem 2 (Chinese remainder theorem)**.** *For any two coprime numbers m and n, picking any two $a \pmod m$, $b \pmod n$ there is a unique $c \pmod{mn}$ such that $c \equiv a \pmod m$ and $c \equiv b \pmod n$.*

The idea is that the Chinese remainder theorem puts in one-to-one correspondence the two congruence classes modulo m and n, on the one hand, and a congruence class modulo mn, on the other hand. However, its proof is a constructive statement dealing with integers : it exhibits the necessary $(c \pmod{mn})$, and the reader is left to deduce the set-lifted link with congruence classes.

Afterwards, it suffices to remark that $\Phi(k)$ is, by definition, the number of congruence classes modulo k, and the equality (2.3) follows. This remark was first made by Gauss,[22] and was reflected in a preliminary proof of equality (2.3) in SSReflect, made by Laurent Théry before our own work on the subject. This last proof even has an ancestor in the proof of correctness of the RSA algorithm included in the contributions to Coq , also based on the Chinese remainder theorem. It is interesting to note that this range of tools ("manually" counting congruence classes, and elementary properties of divisibility) was also the state of the art previously employed for formalized reasoning on Euler's totient function (Fujisawa et al. 1998).

[22] *Disquisitiones Arithmeticae* (1801), art. 38.

Our goal was to equip the SSReflect library with enough abstract tools to reach the modern (and much simpler) statement of the proof:

Proof : Theorem 1 on p. 93. Interpret as equality in the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$, and apply Lagrange's theorem[23] saying that the order of a finite group "kills" all its elements. □

[23] Lagrange's theorem *per se* relates the order (number of elements) of subgroups with their parent:

Theorem 3 (Lagrange)**.** *The order of any subgroup* H *of* G *divides the order of* G.

However, a simple consequence of the theorem is that the *order* p of any element a of a finite group G (p is the smallest k with $a^k = 1$) divides the order |G| of the group, since p is also, by definition, the order of the cyclic subgroup generated by a. Therefore $a^{|G|} = 1$.
We use × (in regular script) to denote the *exterior* direct product, *i.e.* the canonical group structure defined on the cartesian product of two groups.

Meanwhile, the modern proof of equation (2.3) still considers the Chinese remainder theorem, but *set-lifted*, and stated this time as a group isomorphism, something that did not occur in mathematical literature before the 1920s (Stillwell 1994, p.107). It is then stated as such:

Theorem 4 (Chinese remainder theorem)**.** *If m and n are coprime, then the map $\rho : r \mapsto (r \pmod m, r \pmod n)$ is a group isomorphism of $(\mathbb{Z}/mn\mathbb{Z})^\times$ onto $(\mathbb{Z}/m\mathbb{Z})^\times \times (\mathbb{Z}/n\mathbb{Z})^\times$.*

The key benefit of the group isomorphism formulation here, is that starting from a value for the inverse of the morphism ρ provided by Theorem 2, and without considering the explicit extensional definition of congruence classes, we can get Equation 2.3 on p. 93 using a simple cardinality lemma.

Hence, with the hindsight on the historical progression of this proof, and given approaches to the formalization, it seems clear that **the notion of multiplicative group modulo n is the key to the simplification of the proofs of Euler's totient function properties.**[24]

[24] The underlying set bears the group structure traditionally associated to group of units $\mathbb{F}_p^*$, the $(p-1)$ multiplicatively invertible elements (*units*) of the field $\mathbb{F}_p$ of characteristic p. We will skew our notations and use of variables closer to that ring-theoretic vocabulary.

2.2.2 *The multiplicative group* $(\mathbb{Z}/p\mathbb{Z})^\times$

The first step towards defining $(\mathbb{Z}/p\mathbb{Z})^\times$ its *group domain* type. For this, we may look at the already defined *additive* modular group. Its carrier type has elements in `ordinal n`, noted `'I_n`:

```
Inductive ordinal (n : nat) := Ordinal m of m < n.

Canonical ordinal_subType :=
  [subType for nat_of_ord by ordinal_rect].
```

The symbol < here stands for the boolean strict order on natural numbers. It seems that all that is left is to define a similar subtype for the multiplicatively invertible elements of `ordinal n`.

At reading this, the reader may have the impression that we are violating our own paradigm of *sets as a selection*: should not the integers up to our modulo be members of a *more general* type ?

The answer is negative, because having the modular addition defined on the set of all integers would require us to declare a potential infinity of `finGroupType` instances on `nat` (of which only one would be inferrable), not to mention the added complexity for defining the modular operations. It is secondly that from the subtyping point of view, those modular integers are in a larger type *already* : the property attached to an ordinal such as declared above is boolean, and proof-irrelevant, so that we can use our `subType` (§ 2.1.2 on p. 86) structure to ensure that the decidable Σ-type this `ordinal n` forms coerces *injectively* to `nat`.

This ease in defining types by *decidable specialization* hints at an easy definition for the group of units modulo p. Given a the boolean predicate `coprime`, we simply define:

```
Inductive Zp_unit := ZpUnit x of coprime p x.
Implicit Types u v : Zp_unit.
Coercion Zp_unit_val u := let: ZpUnit x _ := u in x.
Canonical Structure Zp_unit_subType := SubType Zp_unit_val
    Zp_unit_rect vrefl.
```

The group we are interested in will indeed be a selection of elements. It will just have the uncommon property of containing all the elements of its underlying `fingroupType` carrier. Thanks to subtypes, we have the ability to quickly see those elements as modular integers, or simply integers, so that the definition remains tractable nonetheless.

The key objects for consideration in our newly defined group type then come from passing around coprimality proofs, as witnesses of membership in the subtype:

```
Lemma Zp_unit_1_proof : coprime p Zp1.
Lemma Zp_unit_mul_proof : ∀u v, coprime p (Zp_mul u v).
Lemma Zp_unit_inv_proof : ∀u, coprime p (Zp_inv u).

Definition Zp_unit_1 := ZpUnit Zp_unit_1_proof.
Definition Zp_unit_inv u := ZpUnit (Zp_unit_inv_proof u).
Definition Zp_unit_mul u v := ZpUnit (Zp_unit_mul_proof u v).
```

```
Lemma Zp_unit_mul1g :
  left_unit Zp_unit_1 Zp_unit_mul.
Lemma Zp_unit_mulVg :
  left_inverse Zp_unit_1 Zp_unit_inv
     Zp_unit_mul.
Lemma Zp_unit_mulA :
  associative Zp_unit_mul.

Definition
     Zp_unit_baseFinGroupType_def :=
  Eval hnf in [baseFinGroupType of
     Zp_unit by
    Zp_unit_mulA, Zp_unit_mul1g &
     Zp_unit_mulVg].
```

Figure 2.12: A `basefinGroupType` for the multiplicative group modulo p

Once this is done, we can define the base group type structure as in Fig. 2.12. The group itself is defined as usual from the axioms above, so that here and in general, strict prefixing by `ZpUnit` should be enough to help the reader see what we are exactly referring to in subsequent lemmas.

In sum, the multiplicative group is defined on the unambiguous set obtained by the intersection of two decidable subtypes:
- the `ordinal` set of integers smaller that a limit `p`,
- and the set of integers coprime to that limit.

Since the *subtype* structure gives them different types for each limit, this provides us with the opportunity to bestow different group instances for each of those sets.

2.2.3 Isomorphisms and morphisms

In this subsection, the word *function* will always stand for a COQ term, never for the mathematical notion. Conversely, the word *mapping* will always designate the mathematical notion of function, with specific domain and range.

The next step towards a simple formalization of the correctness of RSA is a description of the set-lifted Chinese lemma mentioned in § 2.2.1 on p. 95. We recall that it states that the the map $\rho : r \mapsto (r \pmod m, r \pmod n)$ is a group isomorphism of $(\mathbb{Z}/mn\mathbb{Z})^{\times}$ onto $(\mathbb{Z}/m\mathbb{Z})^{\times} \times (\mathbb{Z}/n\mathbb{Z})^{\times}$. We therefore need to define group morphisms. Since we are in a finite context, all we will need from there to isomorphisms is an injectivity property.

We start with the mathematical definition of an homomorphism:[25]

Definition 4 (Dummit and Foote 2004)**.** *Let (G, ☆) and (H, ◇) be groups. A map* $\varphi : \mathrm{G} \to \mathrm{H}$ *such that:*

$$\varphi(x \star y) = \varphi(x) \diamond \varphi(y) \text{ for all } x, y \in \mathrm{G}$$

is called a homomorphism.

Translating that definition in COQ will put our definition of *sets as a selection of elements* of a given type under the spotlight. The mathematical mapping naturally comes with a proper domain. This is equally implicit when the COQ equivalent of that mapping is a (necessarily total) COQ function whose carefully-crafted *type* represents its domain.

Schematically, we can envision our definition of sets (Fig. 2.8 on p. 90) as a list l of elements of type T — modulo a helpful structure package. Yet we still want to avail ourselves of the function-based terms of COQ, and therefore want a function to be involved in the representation of a mathematical "mapping with domain", even if the underlying object we choose to represent it is not.[26] Therefore, the question is: to represent a mapping of domain `l1 : list T1` and of range `l2 : list T2`, which function of — necessary — type `(T1 → T2)` should we choose?[27]

Whichever function `(f : T1 → T2)` we adopt to represent a mapping $\varphi : \mathrm{G} \mapsto \mathrm{H}$, it seems clear that it will have to be mimic the behavior of φ within its domain. The difficult issue is to deal with the values of `f` outside the `finset` that will represent G or, equivalently, that there is an infinity of COQ functions that behave like φ within said representant. In most part to ease dealings with morphism composition, we adopted a definition that selects one particular representant of this set of candidate functions.

We explain it without further discussion, since we will come back to an analysis of this definition under a different light in § 2.3.1 on p. 106: **the gist of it is that the existence of a unit for groups permits a "dummy return" approach**. Though the fundamental difficulty here is in representing a *mapping-with domain*, not necessarily a *morphism*, the addition of the single, localized morphism property exemplifies techniques we explained

[25] An homomorphism must preserve structure, here that of a group. The unit's u_G image is u_H by applying the definition's property to $u_G \star x$, $x \star u_G$ and recalling group units are unique. The image of the inverse is computed by applying the definition's property to $x \star x^{-1}$.

[26] As with the finite mappings of § 2.1.4 on p. 89, internally represented with lists of elements of the range indexed by the enumeration of the domain — but still coercing to COQ functions.

[27] We mean list in the abstract sense, here, see note 33 on p. 35.

There is a reason we want to use COQ's functions to represent morphisms rather ad-hoc tuples inspired from those we have used to define finite functions in § 2.1.4 on p. 89, but defined — without serious difficulty — in a partial manner: functions have native identity and associativity laws we can benefit from. Those laws correspond to the functoriality of the lookup tables of those partial-finite "functions", but this is not accessible directly in-calculus, something we will come back to in § 3.4.4 on p. 134.

in the previous chapter without clouding the issue too much. We go back henceforth to defining morphisms.

Suppose we want to build the concrete representation in COQ of a morphism φ defined on a domain G. The method we describe here was explained in (Gonthier et al. 2007).

The representation of G itself is built as a structured set `G` of elements of a type `gT`, the latter being equipped with the appropriate `Canonical Structure` — that is, an instance of the `finGroupType` record type. A similar instance `hT` of `finGroupType` will represent the type of elements in the range of the function. The user wanting to define φ will then have to define a function `f` as follows:

$$\mathtt{f} : (x : \mathtt{gT}) \mapsto \begin{cases} 1_{\mathtt{hT}} & \text{if } x \notin \mathrm{G} \\ (\varphi(x) : \mathtt{hT}) & \text{otherwise} \end{cases}$$

This defines a morphism with the following parameters: we can compute a *kernel* $\mathcal{K}$,[28]

```
ker f := {x : gT | {∀y : gT, f (x * y) = f y}}.
```

and a *domain* $\mathcal{D}$,

```
mdom f := ker f ∪ {x : gT, f x ≠ 1}.
```

Then, by just providing a proof that $\mathcal{D}$ is a group on which $\forall x, y \in \mathcal{D}$, $\mathtt{f}(xy) = \mathtt{f}(x) \cdot \mathtt{f}(y)$ [29], the user can declare a new instance of `morphism`. Indeed, provided that φ is not the trivial morphism:[30]

Lemma 1. $\exists w, \varphi(w) \neq 1 \Rightarrow \mathrm{G} = \mathcal{D}$

Proof. $\mathrm{G} \subseteq \mathcal{D}$ is clear, since for all x in G, $y \in \mathrm{G} \Leftrightarrow xy \in \mathrm{G}$. For the converse, the difficult case is when $x \in \mathcal{K}$. Then $\mathtt{f}(xx^{-1}w) = \mathtt{f}(w) \neq 1$, while if $x \notin \mathrm{G}$, $\mathtt{f}(xx^{-1}w) = \mathtt{f}(x^{-1}w) = 1$. □

`f` is a representative of the class of functions of type `gT → hT` whose restriction to $\mathcal{D}$ is φ — the example of a frequent pattern in our library: [31] rather than working with all objects that behave well under a local hypothesis, we prefer building a normalized representative that enjoys that property in an unequivocal fashion.

In conclusion, our definition for morphisms is:

```
Variables (elt1 elt2 : finGroupType).
Structure morphism : Type := Morphism {
  mfun :> elt1 → elt2;
  group_set_dom : group_set (dom mfun);
  morphM : ∀x y, dom mfun x → dom mfun y →
           mfun (x * y) = mfun x * mfun y
}.
```

Then, the definition of injectivity is but a simple notation:

```
Notation "''injm' f" := (pred_of_set ('ker f) ⊆ pred_of_set 1)
  (at level 10, f at level 8, format "''injm'  f") : group_scope.
```

2.2.4 Automorphisms

We start with some example morphisms, with the intent of working up to the isomorphism ρ mentioned in § 2.2.1 on p. 95.

The first field-test for the use of isomorphisms is cyclic groups:[32] they are the simplest non-trivial groups, their structure is well-known, so is that

[28] Compare this COQ encoding to the usual mathematical definition:

Definition 5 (Dummit and Foote 2004). *Let* G *and* H *be groups and* $\varphi : \mathrm{G} \mapsto \mathrm{H}$ *be a homomorphism. [We] define the* kernel *to be* $\{g \in \mathrm{G} | \varphi(g) = 1_{\mathrm{H}}\}$.

[29] Something we will call *being morphic* on $\mathcal{D}$

[30] Given a group G, we call the surjective morphism $f : \mathrm{G} \to 1$ *trivial*, or trivialising.

[31] Look *e.g.*, at the definition of quotients in (Gonthier et al. 2007, §3.3).

[32]

Definition 6 (Dummit and Foote 2004,p. 68). *A finite group* H *is cyclic if* H *can be generated by a single element, that is if there is some element* $x \in \mathrm{H}$*, such that* $\mathrm{H} = \{x^k | k \in [0; n], n \in \mathbb{N}\}$ *(where as usual the operation is multiplication).*

[33] The *automorphism* group of G is the group formed by the bijective endomorphisms $\mathrm{G} \to \mathrm{G}$, with function composition as the law and the identity function as trivial element.

of their automorphism group,[33] and both are made clearer by mapping to a finite group corresponding to some initial segment of $\mathbb{N}$.

Any cyclic group of order n is isomorphic to the additive group of integers less than n and this isomorphism is defined in the following way:[34]

[34] We have introduced the notation for ordinals at the beginning of 2.2.2 on p. 96.

Definition 7.
```
Definition Zpm n a (i : 'I_n) := a ^+ i.
```

The declaration of the morphism canonical structure is deduced from the proof of the characteristics properties — since we note the exponentiation induced by the multiplicative group law with the infix ^+:

```
Lemma ZpmM : ∀(n : pos_nat) a, a ^+ n = 1 →
  {in Zp n &, {morph Zpm a : x y / x * y}}.
```

The integer parameter n in this definition constrains the ordinal type, but is never re-used in the actual term (Zp n): it is itself no other than [set x :'I_n|true] — also known as the full underlying domain.

Then all that remains is an injectivity proof which is obtained without difficulty from a cardinality argument:

```
Lemma Zpm_inj : ∀a, injective (@Zpm #[a] a).
```

Likewise, note this injectivity property depends on the order #[a] defined literally as the smallest integer such that a^n = 1+:

```
Notation "#[ x ]"  := (order x) (at level 0, format "#[ x ]") :
    group_scope.
```

As we have seen in note 23 on p. 95 there are two definitions for the word "order" in group theory: the one just above, and the cardinal of a group.

Cyclic groups is where those definitions meet : a cyclic group is defined as the set generated by a single element x — and the generated set is obtained by integrated intersections with a method depicted by Bertot et al. (2008).

```
Definition generated A := \bigcap_(G : groupT | A ⊆ G) G.
Definition cycle x := generated [set x].
```

Then the definition of order of an element becomes, as expected (see Fig. 2.6 on p. 89 for notations) :

```
Definition order x := #|cycle x|.
```

This is the simplest isomorphism one can define, and usually the first encountered in an undergraduate course. The second one can expect is the correspondence between the automorphism group of a cyclic group of order n, and our now familiar multiplicative group modulo n.

Automorphisms, however, have yet to be defined. And our policy of defining sets as a selection of elements means that this definition might be slightly more complex than usual.

Automorphisms are bijective endomorphisms. Hence, they have to relate somehow to functions of type gT→gT, for gT a finGroupType. But gT → gT is not a finGroupType itself: *only some of its elements are invertible.*

We have to define a group type for invertible functions (that is, in a finite context, injective functions) of a group type unto itself: In sum, a type of permutations. Thankfully, they are the selection of those functions of type gT → gT for which a decidable injectivity property is true: hence, they can be defined with a subtype, much in the way we dealt with the multiplicative group in § 2.2.2 on p. 96.

```
Inductive perm_type : predArgType :=
  Perm (pval : {ffun T → T}) & injectiveb pval.
Definition pval p := let: Perm f _ := p in f.
Definition perm_of of phant T := perm_type.
Identity Coercion type_of_perm : perm_of ↦ perm_type.

Notation pT := (perm_of (Phant T)).

Canonical perm_subType :=
  Eval hnf in [subType for pval by perm_type_rect].
```

Figure 2.13: Definition of a type of permutations

The definition for permutations is given in Fig. 2.13. Note the use of our finite functions, ensuring that our type of permutations coerces to the underlying Coq functions type. It also features a phantom type argument for the benefit of notations, as explained in § 1.4.3 on p. 70.

From this type of permutations, automorphisms are nothing but the selection of those injective endomorphisms that are also surjective and enjoy the characteristic property of morphisms on a group subset of their domain type:

```
Variable gT : finGroupType.
Implicit Type A : {set gT}.
Implicit Types f g : {perm gT}.

Definition Aut A := [set f | perm_on A f && morphic A f].
```

Since, however, the fact that `A` is a group in the definition above is only an auxiliary expectation that is not necessary for the definition of automorphisms, we do not require it at this stage (`A` is a set above). It will be easy to add to the context of lemmas that might make use of it along the library we develop for automorphisms and more importantly, the presence of the simplest possible type as an argument of the automorphism constructor ensures *projections* from group types will be inserted *as coercions* at the appropriate positions : the application of our automorphism lemmas will trigger group inference if necessary.

One remarkable consequence of that definition is that automorphisms are *not* morphisms in the sense defined above : the permutation type requires they return *distinct* values for any pair of elements in their domain type and *not only on the subset of which they are an isomorphism*. Nonetheless, it was still possible to define a notion of automorphism whose domain was determined by its values, using the same trick as in the previous section: we made automorphisms behave as the identity outside of their domain.

We will come back on how we fared with that definition. For now, we go back to *isomorphisms*.

2.2.5 *Cycles and integers modulo n*

The first isomorphism we have seen above (Def. 7 on the previous page) also defines the additive action of the generator a on its "own" cyclic group.[35]

The idea of the isomorphism between the group of automorphisms of a cyclic group and $\mathbb{Z}/n\mathbb{Z}^{\times}$ is that any other *generator* of the cyclic group `<[a]>` would give rise to an action that would also generate the whole group.

If we define the action obtained from any element of the group, we write:

```
Variable a : gT.
```

[35]

Definition 8 (Dummit and Foote 2004, p.41). *A* group action *of a group* G *on a set* A *is a map from* G*x*A *to* A *(written as* $g - a$*, for all* $g \in G$ *and* $a\,in\,A$*) satisfying the following properties:*

1. $g_1.(g_2.a) = (g_1 g_2).a$*, for all* $g_1, g_2 \in G, a \in A$*, and*
2. $1.a = a$*, for all* $a \in A$.

```
Variable n : nat.

Definition cyclem of gT := fun x : gT => x ^+ n.
```

However, it is not yet clear that this function returns an automorphism, in the sense of our definition above. To ensure that, we have to prove that if its first argument is a member of $\mathbb{Z}/n\mathbb{Z}^\times$, we get an injective function. The proof is straightforward and is followed by the demonstration that we, indeed, generate the whole group:

```
Variable u : Zp_unit #[a].

Lemma injm_cyclem : 'injm (cyclem u a).
Proof.
apply/subsetP=> x; case/setIdP=> ax; rewrite !inE -order_dvdn.
rewrite -order_eq1 -dvdn1; move/eqnP: (valP u) => /= <-.
by rewrite dvdn_gcd order_dvdG.
Qed.

Lemma cyclem_dom : cyclem u a @@* <[a]> = <[a]>.
Proof.
apply/morphim_fixP=> //; first exact: injm_cyclem.
by rewrite morphim_cycle ?cycle_id ?cycleX.
Qed.
```

From those proofs, we have defined a construction that builds the automorphism which corresponds to our original COQ function (and coerces to it).

```
Definition Zp_unitm := aut injm_cyclem cyclem_dom.
```

It is that final function that we show to be an isomorphism in a manner similar to what was done above for the additive case.

```
Lemma Zp_unitmM : {in Zp_units #[a] &, {morph Zp_unitm : u v / u
    * v}}.
<...>
Lemma injm_Zp_unitm : 'injm Zp_unitm.
<...>
```

This isomorphism made it elementary to prove that the automorphism group of a cyclic group is abelian, for instance:

```
Lemma Aut_cycle_abelian : abelian (Aut <[a]>).
Proof. by rewrite -morphim_Zp_unitm morphim_abelian
    ?Zp_units_abelian. Qed.
```

And moreover, it is establishing a link between Φ and the generators of a cyclic group: knowing that a^k is another generator of the cyclic group generated by a if and only if k is coprime to the order n of a, it should now be clear that:

```
Lemma phi_gen : phi #[a] = #|[set x | generator <[a]> x]|.
Proof.
<...>
Qed.
```

To continue on Φ, it was finally easy to provide a definition for the isomorphism ρ (see definition in Thm. 4 on p. 95) of type

```
Zp_units (m * n) → (Zp_units m) * (Zp_units n)
```

through injections within the subtype of the multiplicative group defined in § 2.2.2 on p. 96. The technique is similar to the `autm` function above.

The injectivity proof was carried out by providing an inverse to ρ, using this time a *number theoretic* Chinese lemma. It led us, finally, to the proof of equation 2.3 on p. 93 above:

```
Lemma phi_mul: ∀m n,
  coprime m n → phi (m * n) = phi m * phi n.
Proof.
move=> m n Hcop.
case: (posnP m) => [→ | mpos]; first by rewrite mul0n phi0
    mul0n.
case: (posnP n) => [→ | npos]; first by rewrite muln0 phi0
    muln0.
have:= @rho_isom (PosNat mpos) (PosNat npos) Hcop.
by move/isom_card; rewrite !cardsT card_prod !cardphi.
Qed.
```

Since we have a cardinality library on injective morphisms, this proof of `phi_mul` was but four lines, down from the original 52.

Euler's theorem (Thm. 1 on p. 93) is then obtained as a simple corollary of the second isomorphism presented above.[36] Its proof, Fig. 2.14 consists in using Legendre's theorem `order_dvdn` followed by a cardinality lemma.

[36] That ultimate step is the work of Laurent Théry.

```
Theorem Euler: ∀a n: pos_nat, coprime a n → a ^ phi n  = 1
    %[mod n ].
Proof.
move=> a n Cop.
have Ha': coprime n (inZp (valP n) a) by rewrite coprime_sym
    coprime_modl.
have Hp1: (ZpUnit Ha') ^+ (phi n) = 1.
  apply/eqP; rewrite -order_dvdn -card_Zp_units.
  by apply: cardSg; rewrite cycle_subG inE.
move/val_eqP: (@Zp_units_expgn n (ZpUnit Ha') (phi n)).
by rewrite /= modn_exp Hp1; move/eqP←.
Qed.
```

Figure 2.14: Proof of Euler's theorem

We now have all the prerequisites of a swift proof of the correctness of RSA.

2.2.6 *RSA's correctness*

Naturally, having equation 2.3 on p. 93 as a library lemma (`phi_coprime`) helps **tremendously**. Hence, to pick e, d verifying the equation 2.2 on p. 93 above, all that remains to do is to pick a random e coprime to $\Phi(n)$ such that we can use Euclid's algorithm to compute d, its multiplicative inverse modulo $\Phi(n)$. The details of those first definitions are represented on Fig. 2.15.

In this introduction of basic lemmas, we noticed that the value of Φ for a prime wasn't included in the library. However, the `Search` tool of SSReflect (Gonthier et al. 2008, §9) allowed us to quickly find a more general lemma on the value of Φ based on the prime decomposition of a number (`phi_pfactor` above).

```
Variable p q : nat.
Variable prime_p : prime p.
Variable prime_q : prime q.
Variable neq_pq: p ≠ q.

Local Notation n := (p * q).

Hypothesis big_p: 2 < p.
Hypothesis big_q: 2 < q.

(* We compute the totient of that product. *)

Lemma pq_coprime: coprime p q.
Proof. by rewrite prime_coprime // dvdn_prime2
    //. Qed.
```

```
Lemma phi_prime : ∀x, prime x → phi x = x.-1.
Proof.
move=> x; move/phi_pfactor; move/(_ _ (ltn0Sn
    0)).
by rewrite expn1 expn0 muln1.
Qed.

Lemma pq_phi: phi(n) = p.-1 * q.-1.
Proof.
rewrite phi_coprime; last by rewrite pq_coprime.
by rewrite !phi_prime //.
Qed.
```

Figure 2.15: First definitions for RSA correctness

The bane of working with a proof assistant is that we can't ignore an often-overlooked detail: if M and n are not coprime (and if $M \neq n$, otherwise this

is trivially true), we can assume, without loss of generality that $p|M$ and q and M are coprime. This case is treated in Fig. 2.16

```
Lemma notcoprime: ∀x, 0 < x < n → ~~ (coprime x n) →
  ((p %| x) && coprime x q) || ((q %| x) && coprime x p).
Proof.
move=> x; case/andP=>[x_gt0 x_lt_n]; rewrite coprime_mulr;
    move/nandP.
move/orP; rewrite coprime_sym [coprime _ q]coprime_sym
    2?prime_coprime //.
case Hdvdn: (p %| x) =>/= ; rewrite ?¬K ?andbF ?andbT //
    orbF=>_.
have xdpp_x: (x %/p) * p = x by move: (Hdvdn); rewrite dvdn_eq;
    move/eqP.
rewrite -xdpp_x; apply/negP=> H; move:H; rewrite (euclid _ _
    prime_q).
rewrite [ _ %| p]dvdn_prime2 // eq_sym; move/¬TE: neq_pq=>↦.
rewrite orbF=>H; move: (dvdn_mul (dvdnn p) H).
rewrite [_ * (_ %/ _)]mulnC xdpp_x; move/(dvdn_leq x_gt0).
by rewrite leqNgt x_lt_n.
Qed.
```

Figure 2.16: RSA:the non-coprime case

In that case $M^{k\Phi(n)}M \equiv M \pmod{q}$ is a consequence of Euler's theorem, and $M^{k\Phi(n)}M \equiv M \pmod{p}$ is trivially true, which allows us to conclude using a Chinese lemma. Most presentations of RSA cop out of treating this case, but it makes the encoding no less correct. We can now prove the theorem:

```
Theorem rsa_correct1 :
 ∀w : nat, w ≤ n → (decrypt (encrypt w)) = w %[mod n].
Proof.
move=> w w_leq_n; rewrite modn_mod modn_exp -expn_mulr.
```

This simplifies the goal to `w ^ (e * d) = w %[mod n]`. Then we are going to want to treat the case where `w = 0` first. Since this is a decidable property, we proceed by making a `case` on `0 < w`. We simplify the left part:

```
case w_gt0: (0 < w); last first.
  move/¬T: w_gt0; rewrite lt0n; move/negPn; move/eqP=>↦.
```

We are brought to `0 ^ (e * d) = 0 %[mod n]`. The left part forces us to check that `0 < e*d`:

```
have ed_gt0: 0 < e * d.
  have:= (divn_eq (e*d) (phi n)); rewrite ed_1_mod_phin => →.
```

We are left with `0 < (e * d) %/ phi n * phi n` 1 %% phi n+. We would like to simplify 1 (mod $\Phi(n)$), which requires $1 < \Phi(n)$:

```
have phi_gt1 : 1 < (phi n); first rewrite pq_phi -(muln1 1).
  by apply: ltn_mul; rewrite -ltnS prednK ?big_p ?big_q
    ?prime_gt0.
```

We can then conclude on both `0 < e * d` and the original case where `w = 0`:

```
  by rewrite [_ * phi n]mulnC (modn_small phi_gt1) addnS lt0n.
by rewrite exp0n ?ltn_addl // ?modn_small.
```

Back to $w > 0$! Simplifying our goal a little further:

```
have:= (divn_eq (e*d) (phi n)); rewrite ed_1_mod_phin modn_small
    // =↦.
```

We get : `w ^ ((e * d) %/ phi n * phi n 1)` = w %[mod n]+, and we are ready to conclude for the case where message and modulus are coprime! **This is the core of the proof** : a simple use of Euler's theorem and a few helper lemma on moduli.

```
case cp_w_n : (coprime w n).
  rewrite expn_add [_ * phi n]mulnC expn_mulr -modn_mul2m; move:
    cp_w_n.
  by move/Euler=>E; rewrite -modn_exp E modn_exp exp1n
    modn_mul2m mul1n.
```

Now, in the case where modulus and message are *not* coprime, we are going to want to use the `notcoprime` lemma above, for which we also require them not to be equal, so we quickly eliminate that case.

```
case w_eq_n: (w == n).
   by move/eqP: w_eq_n =>->; rewrite -modn_exp modnn exp0n ?mod0n
    ?addn1.
```

Finally, we apply the Chinese lemma, to separate our requirements on $w \pmod{\Phi(n)}$ into requirements on $w \pmod{\Phi(p)}$ and $w \pmod{\Phi(q)}$:

```
move: w_leq_n; rewrite leq_eqVlt {}w_eq_n orFb; move=> w_lt_n.
apply/eqP; rewrite chinese_remainder; last first.
  by rewrite prime_coprime // dvdn_prime2.
```

The goal at this stage is

```
(w ^ ((e * d) %/ phi n * phi n + 1) == w %[mod p]) &&
  (w ^ ((e * d) %/ phi n * phi n + 1) == w %[mod q])
```

We want to rewrite this a little further to make the prime w is *not* divisible by more prominent.

```
rewrite mulnC {1 3}pq_phi {2}[_ * q.-1]mulnC -2!mulnA 2!expn_add
    expn_mulr.
rewrite [w ^ ( q.-1 * _ )]expn_mulr expn1 -modn_mul2m
    -(modn_mul2m _ _ q).
rewrite -modn_exp -(modn_exp _ q); move/andP: (conj (idP w_gt0)
    w_lt_n).
```

This gives us the goal:

```
((w ^ p.-1 %% p) ^ (q.-1 * ((e * d) %/ phi n)) %% p * (w %%
p) == w %[mod p]) &&
  ((w ^ q.-1 %% q) ^ (p.-1 * ((e * d) %/ phi n)) %% q * (w
%% q) == w %[mod q])
```

We can now use `notcoprime` above, apply Euler's theorem in each case, and conclude:

```
move/andP: (conj (idP w_gt0) w_lt_n).
move/notcoprime; move/(_ (¬T cp_w_n)); case/orP;
 move/andP=> [Hdvdn Hncp].
   move: Hdvdn; rewrite /dvdn; move/eqP=>->; rewrite muln0
 mod0n /= .
   move/Euler:Hncp; rewrite (phi_prime prime_q)=>->.
   by rewrite modn_exp exp1n modn_mul2m mul1n.
move: Hdvdn; rewrite /dvdn; move/eqP=>->; rewrite muln0 mod0n
 /= andbT.
move/Euler:Hncp; rewrite (phi_prime prime_p)=>->.
by rewrite modn_exp exp1n modn_mul2m mul1n.
Qed.
```

This concludes our certification of RSA's correctness, **in itself 30 lines of proof script**. Note we have never used any automated tactic database such as `auto`, which confers our proof script a particularly deterministic nature (but it's not a requirement, of course). Here's a run of `coqwc`, the COQ line-count utility, on our final file:

The code for our development on Φ, cyclic groups and automorphisms has seen several evolutions since then, most notably the re-working of the definitions of morphism and automorphism from a global to a local version. We are about to treat it in detail. This made the set-lifted Chinese lemma disappear — initially only in a transitory fashion, so as not to depend on soon-to-be-improved definitions, then in a more permanent manner when `phi_mul` was re-worked into a number-theoretic proof, before the re-insertion of the new definition of morphisms.

However, the definition and library for automorphisms, any place where cyclic groups use the notion of generated group, and the isomorphisms with the additive and multiplicative groups modulo n are directly issued from the work of the author as presented above.

Moreover, our re-work of the RSA contribution has been independently developed on top of SSReflect release 1.3pl1, and is available on Github: https://github.com/huitseeker/thesis-spikes/tree/RSA

```
     spec    proof comments
       28       53       17         RSA.v
```

The final file is 126 lines.

2.3 Morphisms and partial functions

2.3.1 Discussion

The model of morphisms presented in § 2.2.3 on p. 97 allowed the Mathematical Components team to formalize a consequent body of algebraic results (Gonthier et al. 2007). It had, by then, been also adopted in other formal developments of algebra (Santen 1999), and featured as a major alternative in surveys of ways to deal with partiality in proof assistants equipped with a logic of partial functions (Müller and Slind 1997). However, further experience with this encoding made us reconsider. We now expose some of its consequences in the hope to inform future designs

A computational domain One objective of this representation was to mimick common practice in mathematics, and have an *implicit* notion of domain that did not need to be passed along with the function — a tedious task faced by previous formalizations dealing with typed sets (Bailey 1998a; Gunter 1989).

Notice that the kernel $\mathcal{K}$ and domain $\mathcal{D}$ (§ 2.2.3 on p. 97) were defined for all functions `f`, whether representing a morphism or not — $\mathcal{K}$ even bore a group structure in all cases. In that regard, those constructions fitted well with our custom of defining objects strictly computationally, and only adding hypotheses on their structure when required by a specific proof. We could always assign a "domain" group to a Coq function `f` (in fact, the largest group on which it could be proven to be a morphism), and the kernel was a group in all cases.

Trouble with the trivial morphism Gonthier et al. (2007) mentioned other advantages to this representation, notably the fact that the composition of two (Coq) functions that were instances of `morphism` was one also. Mathematically speaking, if ψ and φ are group morphisms, $\psi \circ \varphi$ is indeed a morphism whose domain is $\varphi^{-1}(\text{dom}(\psi))$: not having to explicitly mention this domain construction in theorems where compositions occur frequently (such as the isomorphism theorems) was quite convenient.

However, the domain we defined on a compound function only made sense when it corresponded to *non-trivial* components. We made this clearer by proving the following in Coq:

Lemma 2. $g \circ f$ *non-trivial* $\Rightarrow \mathcal{D}(g \circ f) = f^{-1}(\mathcal{D}(g)) \cap \mathcal{D}(f)$

The form of this helper lemma was in fact characteristic of most results on morphisms in our library. Since, as shown in lemma 1 on p. 98, the connection with the usual notions of domain and kernel (and of preimage, etc) was imperfect, we provided the classic results about those notions under a *non-triviality* hypothesis: our careful design just exchanged the hinderance of one local context (making sure the argument of the morphism was in the domain) for another (making sure the morphism was non-trivial) — albeit lighter.

More importantly, though, co-developers of our library knew simple textbook results on morphisms intimately, so they were drawn to prove

theorems about them in two parts: they provided a disjunction that allowed them to expedite the trivial case, then they went on with the proof in a context where all the usual results held as expected. Since this happened more than necessary, the "psychological conditioning" was a disadvantage, even though the overhead involved was often not crippling in terms of script lines.

Morphism restrictions and automorphisms If we follow our idea by which an instance of the `morphism` structure is a morphism that encodes its domain *in its values*, then we should produce one new COQ function for each restriction we make of the domain of said morphism. And we did — around the release of SSReflect 1.1 — but surprisingly, this did not *need* to be explicit in our library until we started to reason about *automorphisms*.

Indeed, most properties we wanted to express on morphisms were set-theoretic results about the (pre-)image of an element or a set, so it was often as easy — though less elegant — to inherit them *on a restriction* by proving carefully stated lemmas on the subsets and elements of those (pre-)images.

The definition of automorphisms (§ 2.2.4 on p. 98) of a group `G:{group gT}`, on the other hand, imposed its own representation requirements:

- they were *endo*-morphisms, and as such had to correspond somehow to functions of type `(gT → gT)`,
- they were members of the automorphism and permutation groups of `G`, hence group elements, so their type had to be of an instance of some `finGroupType` that coerced to `gT → gT`. In particular, they all had to have have an *inverse* by its group law. Since we needed the law for automorphisms to be function composition, it meant this `finGroupType` would only contain invertible, *i.e.*, totally injective functions of type `gT → gT`.

To sum up, so as to keep types as the carriers of group laws, we had to give automorphisms a different semantics from the one we gave to morphisms.

Note that even though `perm gT` was a subtype of `gT → gT`, and elements of the first coerced injectively to the second, coerced automorphisms were of no avail to us: they only coincided with the morphisms we hoped for in case their domain was the set containing the whole of `gT`.

In sum, we could prove `(f in Aut G)` if `f` coerced to a function behaving like *the identity* outside of `G`. On the other hand we could put a `morphism` instance on a *function* `f` (i.e. define a record instance of which `f` would be the first projection) only if `f` behaved like *the trivial morphism* outside of `G`. Both could not happen simultaneously.

Since the semantics of this construction went unnoticed by the type system, we initially had to solve this problem by developing a theory of morphism restrictions (Garillot 2008). It connected a free-form function and the proof that it was *morphic*[37] on a group `H` with the corresponding (but usually distinct) normalized `morphism` — formalizing, in effect, the modelization process explained in § 2.2.3 on p. 97. This was enough to inherit *some* local properties of morphisms for automorphisms. Thanks to a careful, separate formulation of the properties brought on by the characteristic property of morphisms (`morphic`) on the one hand, and injectivity on the

[37] The definition was mentioned in the definition of automorphisms, in § 2.2.4 on p. 98.

other hand, this normalization operator on morphisms themselves, gave a meaning to *morphism restrictions* as an added benefit: naturally, the restriction of `f` to a subdomain `C` was the same as "normalizing" `f` seen as a morphism of `C`. The complexity of use of the operator, however, made the process unwieldy.

2.3.2 Directing inference

The key insight that led us to defining partial functions was to notice that the domain of a morphism did not have to be inferred unambiguously from the underlying function. In all previous applications of computational `Canonical Structure` inference, we had the advantage of having a sufficient compositional definition preexist that of the composition of structures: when we are talking about the intersection of two groups $A \cap B$, for instance, the *set* representing the intersection of the *sets* A and B is defined *before* we prove anything on its having a group structure. It is this expression at the *set* level that forces the coercion of *groups* A and B (which is also their first projection) to *sets* to appear within the composed expression.[38] In a second phase, we explain the `Canonical Structure` mechanism *how* to pick up the presence of projections in the *set*-level expression to form a *group* instance. With the notion of *domain of a morphism*, for example, the situation is different: there is not enough computational content in a function to give a clear reference to a set that would later turn out to be a morphism domain. In fact, for a given morphism, unless we use the complex value-encoding explained above, *many* equally valid domains could be defined (for any morphism domain, all its subgroups are also valid candidates).

[38] Consider for example the definition of set union at the end of § 2.1.4 on p. 89

Since there is no reference to a meaningful computational definition that would deal only with the "supertype" of the structure we are dealing with — functions of COQ, in a way similar to the "set" of groups — the naive definition of a *domain* function would not involve coercions, and thus would not insert projections: we are, as explained in (§ 1.3.4 on p. 56), defining a theory for morphism domains naively will not be effectively usable on practical COQ functions, because we do not know how to make COQ see them as (partial) morphisms.

We therefore need a way to define domains computationally by imperatively telling COQ to query the `Canonical Structure` mechanism, retrieve the precise morphism instance we registered, and use if to compute the domain. Therefore, we now focus on how to manipulate `Structure` inference imperatively, before going back, in a second phase, to how we use this to treat morphisms.

Generic Phantom insertion As seen in § 2.3.1 on p. 106, we want to use `Canonical Structure` inference to remember the name and characteristics of morphisms. However, the inference will only work based on a *projection* present in the term *on which basis* we want to prompt the memory of some structure. In this section, we suggest ways to present COQ with a unification problem that forces it to place the structure projection among the arguments of a function such as the domain, by adding *phantom types*.

A phantom type — an appellation coined by Leijen and Meijer (1999)

— is a parameter of a type definition that has no occurence on its right hand side. Phantom types are well-known in functional programming, and are frequently used to refine operations on runtime values of a type by distinguishing subsets of them with a superfluous (i.e. *phantom*) static type.

However, in our dependently typed framework, the value reflection machinery usually developed in most applications of phantom type techniques is unnecessary, and we hope this exposition will serve as an enticing example of how simple our method is.

As we have mentioned (§ 1.3.1 on p. 51), the key to having COQ do the `Canonical Structure` lookup for us is to insert a canonical projection in the arguments of the function. If we are designing an auxiliary function — say, the domain of a morphism — for one specific `Structure` alone, we can do this within the definition of the auxiliary. If our helper function then turns out to effectively use other members of the instance's `Record`, no matter: the projection that has been inserted in its arguments will just fit the definition of a *phantom type*.

It turns out we can adopt a generic pattern for all `Structure`s, in effect the simple `phantom` definition we mentioned more covertly in § 1.4.3 on p. 70:[39]

```
CoInductive phantom (basicT :  Type) (casper : basicT) :Type:=
  Phantom.
```

[39] As usual in SSReflect parlance, the `CoInductive` keyword is just here to avoid the generation of recursors that occurs with `Inductive`s. (Gonthier et al. 2008, §11.1).

This is, in effect, the simplest exposition of a value at the type level we can possibly craft in COQ. Assume now that we have a helper function that takes an argument `mf : morphism`.[40] We can make it take an additional argument `Phantom (aT → rT) mf`, since as planned, COQ will type that last argument as `phantom (aT → rT) (mfun mf)`. This simple manual insertion of the coercion is enough to provide an explicit `Canonical Structure` constraint.

[40] We refer here to the definition in § 2.2.3 on p. 97.

Since `mf` can be found by unification during the typing of `Phantom (aT → rT) mf`, we make it an `Implicit Argument` of the helper function. As a last component of the mechanism, we mask the passing of `Phantom (aT → rT) f` behind a `Notation`, to the effect that the function does not require the user to specify a full `morphism` instance anymore, but rather makes do with just a projection: the insertion of the `phantom` term is entirely structural, provided we know which type of structure (and hence, which projection) our helper function requires.

As we have mentioned often in this document, our concepts correspond more frequently to *type* classes than to *value* classes. We have developed an instance of the "type injector" above that acknowledges that (the instance for `basicT` = `Type`):

```
CoInductive phant (p : Type) :=
Phant.
```

Naturally, this idiom can still be used to recall a complex record parameter to notations. In effect, we are just using the `phantom` definition to push a value at the type level. This can be used to insert it in an expression that contains a record projection in order to set up the unification process at type inference, as we do here for `Canonical Structure`s, to expose a record parameter to notations, or so as to implement Pebble-style parameter sharing, as exposed in § 1.4.3 on p. 70. We now investigate another unification-related use of this multipurpose tool.

Cloning structures Beyond defining functions dependent on `Structure` elements, this pattern evolved to bypass a quirk of `Canonical Structure` inference:[41] since inference is directed by the head constant of a term, and since COQ loathes unfolding definitions, named aliases do not inherit the

[41] The `cloning` idiom is an invention of Georges Gonthier, that phantom types later came to simplify, as exposed in the end of this paragraph.

Structures of the object they are a copy of. For example, in the following declaration, `the_identity` is given a morphism instance `id_morphism`, but this instance can not serve for `id_alias`:

```
Canonical Structure id_morphism := @Morphism the_identity ...
Definition id_alias := the_identity.
```

This would only be a caveat to the programmer if it were easy to do "manual" inheritance: re-declaring the instance on the spot. But as is, this (unjustly) requires remembering all the arguments to be passed to the constructor Morphism of the structure: they should be unifiable to those within the `id_morphism` definition.

To solve this issue, we use custom instance declarations using constructors that trigger the inference of parents.

We slice the constructor in two parts: the *cloner*, and the actual, notation-based application of the structure constructor. We start with a helper function for dependent matches. It returns the type of a constructor, given the constructor's name and one of its applications:

```
Definition argumentType T P & ∀x : T, P x := T.
Definition dependentReturnType T P & ∀x : T, P x := P.
Notation "{ 'type' 'of' c 'for' s }" := (dependentReturnType c s)
  (at level 0, format "{ 'type'  'of'  c  'for'  s }") :
    type_scope.
```

Then the idiom consists in declaring, for each new structure `str` of constructor `Str`, a `clone` function that takes a structure, unpacks it by pattern-matching, and retains the last `(n-1)` arguments of the `n`-argument constructor `Str`. It returns a functional that given a new constructor, applies it to those last arguments.

```
Definition clone_str s u
  let: Str _ x y ... z := s return {type of Str for s} → str in
  fun k => k _ x y ... z.
```

A notation then sets up the application of an η-expanded constructor which has already been applied the alias in the position of first projection:

```
Notation "[ 'str' 'of' T ]" := (clone (fun x => @Str T x))
  (at level 0, format "[ 'str'  'of'  T ]") : form_scope.
```

This solves the δ-reduction problem, but this idiom is however not enough to explain the `clone` functions written in the first chapter of this thesis, say for example in Fig. 1.31 on p. 66. This is because those constructors use the same idea with a distinct implementation based on the presentation of unification problems through phantom types.

The idea is that the use of the identity function can trigger static unification. For example, when we need a structure `sT` over a type `T`, we can take as arguments `T`, `sT`, and a "dummy" function `T → sort sT`,

```
Definition foo T sT & T → sort sT := ...
```

We can then force the phantom term to be the identity by calling `@foo T sT idfun`, where `idfun` is a constant whose definition is the identity function. The `phant_id` type lets us extend this trick to allow *value classes*.

```
Definition idfun T := @id T.
Prenex Implicits idfun.

Definition phant_id T1 T2 v1 v2 := phantom T1 v1 → phantom T2
    v2.
```

Thanks to this definition, we can sidestep dependent type constraints when building explicit records, e.g., given `Record r := R x; y : T(x).` if we need to build an `r` from a given `y0` while inferring some `x0`, such that `y0 : T(x0)`, we pose

```
Definition mk_r .. y .. (x := ...) y' & phant_id y y' := R x y'.
```

Calling `@mk_r .. y0 .. id` will cause Coq to use `y':= y0`, while checking the dependent type constraint `y0 : T(x0)` We make that final call in notations, which now explains the `clone` functions we have explained in previous sections.

Looking up a structure We now know, for any `Structure s`, how to manually get a handle on some `s`-instance, given a term we know it to canonically project to.

Going back to our question of § 2.3.1 on p. 106, given a function `f : aT → rT` on which we know we have declared a `morphism` structure, how do we make COQ return 'the `morphism` instance for `f` ? Or more precisely 'the term of type `morphism` whose canonical projection will be `f`?

We do it by decomposing the two requirements: on the type of the `Structure`, first, and on what its projection should be, second. Let us consider the following definition, fraught with *phantom types*:[42]

[42] As usual in SSReflect parlance, the `CoInductive` keyword is just here to avoid the generation of recursors (Gonthier et al. 2008, §11.1).

```
CoInductive put (basicT structT : Type)
  (somebasic1 somebasic2:basicT) (someinst:structT) :Type := Put.
```

The key element to remember is the coercion we mentioned in § 2.3.1 on p. 106: if `f` is a function with a type suitable for morphisms, COQ types `(fun (g:morphism) => Put f g g)` as `fun (g : morphism) => Put f (mfun g) g`, where `mfun` is the member of the `morphism` record that projects it to a function. This fits our first requirement. We now have to equate such a structure with `f`, and of course, return said structure at some point. Hence we define:

```
Definition get basicT structT
  (mybasic: basicT) (myinstance:structT)
  (x: @put basicT structT mybasic mybasic myinstance) :=
  myinstance.

Notation "[ 'the' structT 'of' basic0 ]" :=
  (get ((fun myinstance : structT => Put basic0 myinstance
    myinstance) _)).
```

Thanks to the long-winded type `put`, only the last argument of `get` is non-implicit. `get` allows us to meet our second requirement, by forcing the hole where we wish to place the coercion of `myinstance` to contain something equal to its first argument — where we will place the projection `f` —, before finally returning the structure we are interested in. The notation ties the knot by building that last argument in a way that makes COQ coerce the next to last occurence of `mystruct`, as discussed above.

We gloss over technicalities of our implementation, here: `get`, for instance, includes an artificial redex so that `[the morphism of f]` is simplified down to the inferred instance as early as possible[43]. In fact, it turns out that this way of doing an explicit lookup in the `Canonical Projections` table of COQ is very useful within proof scripts : it allows to test the `Canonical Structure` mechanism, and to check inferrability of a given structure on

[43] We invite the reader to go through `morphism.v` in the SSREFLECT library for other details, including `Notation` levels and pretty-printing we omitted here.

any given term. That may however be too complex for our morphism representation needs.

2.3.3 *Morphisms with friendly ghosts*

Definitions The latest definition for morphisms in the Feit-Thompson Theorem library is the following:

```
Variables aT rT : finGroupType.

Structure morphism (A : {set aT}) : Type := Morphism {
  mfun :> aT → FinGroup.sort rT;
  _ : {in A &, {morph mfun: x y / x * y}}
}.
```

The domain is now a parameter of the structure. Unfortunately, COQ does not allow several instances of the same `Canonical Structure` to be registered in the `Canonical Projections` table if they project to the same head constant — though those instances can be defined. In concrete terms, that means that while a given function of type `(aT → rT)` can correspond to several instances of `morphism`, only one[44] of them can be inferred automatically. We will come back on how we manage this restriction.

[44] The first the user has declared.

We now can define the domain operation as such:

```
Variables (aT rT : finGroupType) (A : {set aT}).
Implicit Type (f : {morphism A ↦ rT}).

Definition dom f (phantom (aT → rT) f) := A.

Notation "''dom' f" := (dom (Phantom (aT → rT) f)).
```

The final type of `dom` is:

```
∀(aT rT : finGroupType) (D : {set aT}) (f : {morphism D ↦
   rT}),
phantom (aT → rT) f → {set aT}
```

The right hand side of the definition of `dom` uses nothing more than the type of `f`. But the last argument of `dom` inserts a record projection by forcing coercions to match through `phantom`, the type of the morphism with that of its projection. If we require COQ to print the type of that last argument fully, we see the morphism projection `mfun` appear:

```
phantom (FinGroup.arg_sort (FinGroup.base aT) →
   FinGroup.arg_sort
(FinGroup.base rT)) (@mfun aT rT D f)
```

Hence, when a user types `'dom (conjg x)`, with `conjg x` : $y \mapsto y^x$ the conjugation morphism:[45]

[45] The conjugation morphism $conj_x : y \mapsto y^x = x^{-1}yx$ is represented with the higher-order function `conjg : gT → gT → gT`, for `gT` a `finGroupType`. Here, `conjg x` is the morphism.

- Expanding notation in `'dom (conjg x)` yields:

  ```
  dom (_:morphism _) (Phantom (aT → rT) (conjg x))
  ```

- Given the type of `phantom`, COQ then has to solve

  ```
  mfun (?:morphism _) ≡ (conjg _)
  ```

  for `?` by unification.

- `Structure` inference provides `conjgm G` (for some group `G`) as a solution for `?`,

- and `dom` can then return the domain `G` from the definition of `conjgm G`

The first clear advantage of this definition is that, this time, the morphism fits the mathematical notion: if the user has a morphism φ of domain `G` in mind, any total function on the type of the elements of `G`, agreeing with φ on `G`, can be declared as a `morphism G`.

Morphism application, with normalizing definitions Contrarily to our first definition, however, many usual notions based on the image of a morphism can not enjoy a group structure anymore — at least not if we choose the (set-lifted) image by a morphism to be the (set-lifted) application of the underlying function. Indeed, the morphism structure only provides us with a *localised* proof that a function is *morphic* for elements in its domain (the meaning of the notation `{in A &, {morph ... }}`): `f@:H`, the set-lifted image of `H` by `f`, only makes sense when `H` ⊆ `G`.

We have solved this problem in a typical fashion for our library: we now reason on a new operator, the *morphic (pre-)image*, that takes the intersection of the functional (pre-)image with the domain of the morphism.

```
Definition morphim of (Phantom (aT → rT) f) :=
  fun B => f @: (A :&: B).
Definition morphpre of (Phantom (aT → rT) f) :=
  fun C : {set rT} => A :&: f @^-1: C.
Definition ker mph := morphpre mph 1.

Notation "''ker' f" := (ker (Phantom (aT → rT) f))
Notation "f @* H" := (morphim (Phantom (aT → rT) f) H).
Notation "f @*^-1 L" := (morphpre (Phantom (aT → rT) f) L)
```

Note the systematic use of phantoms and `Notation`s to allow us to use those operators by just passing a function rather than its pre-declared morphism. Those definitions allow us to recover all the usual properties of those objects, as we will show in the next few paragraphs.

Morphism restrictions, with yet another phantom As we have seen in previous examples, when looking for some instance of a given `Structure` type, Coq tries to match a projection of that `Structure` with the head constant of a term. Unfortunately, only one instance can be registered per head constant, which means that if we want a function to be understood as the canonical projection of several possible `morphisms`, we have to change the head constant of that term somehow.

We do this using a constructions that adorns the function with a phantom type:

```
Variables aT rT : finGroupType.
Variables A B : {set aT}.
Definition restrm of A ⊆ B := @id (aT → FinGroup.sort rT).
```

This allows us to build a specific instance of morphism on any function for which we can already derive a `morphism` instance:

```
Hypothesis sAB : A ⊆ B.
Variable f : {morphism B ↦ rT}.

Canonical Structure restrm_morphism :=
  @Morphism aT rT A (restrm sAB (mfun f))
    (sub_in2 (subsetP sAB) (morphM f)).
```

`id` is just the polymorphic identity function, being passed the function type expected of any `morphism` projection. The transparent 'mask' of `restrm` can be 'put on' a function by rewriting, since it replaces convertible terms. Then, when COQ tries to find the `morphism` `Structure` on `restrm sAB f`, it will substitute the restricted morphism instance on `A` for the term, passing the constraint on `f` to be a `{morphism B ↦ rT}` to the rest of the unification process.

Usability example : the composition morphism As previously in (Gonthier et al. 2007), function composition has a canonical morphism structure, whose domain we never have to pass as an explicit argument to enjoy morphism properties. More interestingly, though, the first lemmas about that definition of our library show we have managed to define a simpler notion of the image of a morphism, that behaves the way the mathematical notion does.

```
Variables (f : {morphism G ↦ hT}) (g : {morphism H ↦ rT}).
Lemma comp_morphM :
  {in f @*^-1 H &, {morph (mfun g \o mfun f): x y / x * y}}.
Canonical Structure comp_morphism := Morphism comp_morphM.

Lemma morphim_comp : ∀A : {set gT},
  (mfun g \o mfun f) @* A = g @* (f @* A).
Lemma morphpre_comp : ∀C : {set rT},
  (mfun g \o mfun f) @*^-1 C = f @*^-1 (g @*^-1 C).
```

As in (Gonthier et al. 2007) — though that detail went unsaid — the notation '`\o`' hides but a trivial composition function needed to make `Structure`s inferrable on such a term[46], and compliant with our finite function library. It *does* simplifies to nothing more than the functional composition the user expects.

[46] As in the last paragraph, we need to present `Canonical Structure` inference with a suitable head constant.

Automorphisms Finally, the structure we adopted to describe automorphism remained pretty much the same throughout the change in morphism representation we made through our whole archive. In that sense, with the exception of the annotation marked (*), the architecture described in § 2.2.4 on p. 98 remains the same.

Indeed, the main difficulty in developing a theory for automorphism was inheriting properties from morphism theory, and it is easy to see having a set of localised properties for functions representing morphisms achived just that. Contrarily to our previous definition, the function automorphism objects coerced to did enjoy a `morphism` `Structure`.

The fact that automorphisms kept a constraint on their values outside their domain while morphisms did not, on the other hand was not problematic: automorphisms are studied mostly as members of an automorphism group, or on their own right, and rarely is it needed to study them as morphism objects.

2.3.4 *Other applications of phantom types*

In a way, the last part of what we propose is in essence the dual of the solution to the configuration problem exposed by (Kiselyov and Shan 2004),

and solved by directing the creation of functions with phantom parameters, using type class constraints : we, on the other hand, build "type class" constraints using phantom types, through coercion-based record projection insertions.

Phantom types have known some fame in programming languages (Cheney and Hinze 2003; Fluet and Pucella 2005, 2006; Hinze 2003; Kiselyov and Shan 2004), being mostly used to add some type-safe behavior at the boundaries of the type-checker, furnishing a way of dealing with untyped input in an algebraic way. We hope to have shown that this ability to spur the type checker can also be put to use in allowing some notations to be more expressive to the user.

Subfunctors of the identity

3

THE ORIGINAL AIM OF THIS THESIS WAS TO DETERMINE A WAY TO DO PROOFS BY ISOMORPHISM. The remainder of this document explores a way we have found to do that — involving a necessary meandering between distant but nonetheless useful concepts.

A proof by isomorphism tries to express that the validity of a proof does not change if one of its objects is replaced by another in the same equivalence class for the isomorphism relation. However, the central problem with expressing this notion as a "shallow" embedding in COQ lies with the fact that two such objects, when talking about an isomorphism of groups, can have distinct types. Indeed, rewriting a property according to a predefined equivalence relation, in COQ, is not new: it is the job of the `Setoid` mechanism — but this machinery only works when substituting terms of the same type.

Instead of having an out-of-language, meta-theoretic tool to treat isomorphism proofs, we decided to concentrate on the mathematical study of *some* properties which are invariant by isomorphism and to translate this process to the study of a distinct, more amenable construct on the groups they qualify. Those properties are studied in § 3, as is their equivalence with the computation of some group-theoretic constructions. The latter are subgroup-defining functions, whose value on the groups they are passed as an argument isolates a class of groups bestowed with a property. We have exchanged, for a large class of useful group-theoretic properties, the study of a *property on a group* with that of a *subgroup defined in a particular way*. In effect, to prove said property on *any* group amounts to studying an *equality* involving a specific subgroup as *defined by a functor*.

The *equality* of that functorial subgroup with a specific, desired value is a statement that is more amenable to transformation than a non-specific property on groups. In particular, the specific shape of that equality allows us to apply a single lemma to transform it into an equivalent modulo isomorphism — provided the class of groups characterized by the value of that functorial subgroup is indeed isomorphism-invariant. Thankfully, for all those functorially-characterized properties, *they are*.

Armed with this useful — but little-used — piece of mathematics, we can effect a method of proof by isomorphism for a comfortable subset of the properties we need to study in the Feit-Thompson proof. Moreover, those subgroup-defining functions are themselves interesting objects of study — in many cases they are already commonly studied objects in the theory

Another approach consists in twisting the definition of group to make it fit the previous model : defining "groups" — or at least what will figure in the statement of a property at the position where a group can be expected — as an *equivalence class of groups*. This is what we would call the "deep" embedding of proof by isomorphism. The difficulty with this technique consists in identifying properties which *are* conserved by isomorphism and modifying their statement without making them too alien compared to those that do not enjoy this invariance. Moreover, it implies maintaining a complex distinction between those polymorphic classes of groups and the sets they contain, all transparently to the user — who is potentially a mathematician and should not see any apparent difference between the two. Nonetheless this approach has been attempted, once, by Santen (1999), albeit in a library much smaller than ours.

The SSR library prides itself on offering a user interface that is as close as possible to common mathematical language. Moreover, the apparition of —- potentially infinite — classes of groups within our properties would destroy any hope as to their decidability. Not wishing to abandon the keystone of small scale reflection, we decided to come back to an explicit (shallow) rendering of invariance by isomorphism. However, the problem of being able to re-write a property with a polymorphic equivalence remains difficult to deal with within the confines of the type-checker. We have therefore chosen to adopt a slightly different tack to deal with the matter.

of finite groups. They are functorial — that is, they give rise to a unique functor of which they provide the object mapping. Their values on any class of groups also, conversely, provide hints as to a group theoretic property. In addition to studying properties using functors, we can therefore study functorial constructions using properties.

Finally, identifying those subgroup-defining functions can be done by simply examining their type. They correspond to polymorphic subfunctors of the identity on the category of groups and their functoriality condition (that *correspondence*) is precisely expressed as a "free theorem" of parametricity. Thus, provided we can have a parametricity result strong enough to obtain this "free theorem" within COQ, we can solve a problem that lingered in all methods representing proofs by isomorphism : how to identify, simply by their statement, those proofs that are invariant by isomorphism. Our solution consists simply in defining them as a compound of *group classes* — that is, properties that can be reflected to functors. Indeed, those functors have, thanks to canonical structures, composition properties that can let us scale their "equivalence with properties" up to complex statements.

In conclusion, the next few sections gradually build the elements of that solution. In § 3 on the previous page, we start defining subgroup-defining functions, their functoriality and their elementary properties. We obtain a method of proving characteristicity on complex subgroups as a side-effect. We move on to define a torsion theory for groups, yielding the desired equivalence between properties characterizing a torsion (or torsion-free) class of groups, and the value of such a functorial on elements of the class. In § 3.4 on p. 131, we attach ourselves to obtaining functoriality properties on our subgroup-defining functions by examining their type. That is, we suggest a method towards obtaining a "free theorem", given a suitable polymorphic constructor. In effect, this amount to the automatic generation of proofs of instances of a relational parametricity theorem. In effect, we describe a method for reifying those COQ terms that correspond to terms of a polymorphic λ-calculus. Then, we use a deep embedding of such a calculus — on which a relational parametricity theorem is proven — to obtain the proof of the free theorem. The combination of those approaches should yield a pleasant way of dealing with some "free theorems" — and, by extension, with a large class of proofs invariant by isomorphism — to the user.

3.1 *Subgroup-defining functions, generalizations, propositions*

3.1.1 *Characteristic subgroups and subgroup-defining functions*

NORMAL SUBGROUPS[1] are a fundamental notion in group theory: a normal subgroup exactly captures the kernel of a congruence relation compatible with the group law it shares with its parent (Lidl and Pilz 1998, p. 103), and thus naturally forms a quotient by this relation.

Proofs of normality abound in the SSReflect libraries, but their construction is somewhat inconvenienced by the fact that the 'normal' relation, noted with the infix symbol $\lhd$, is not transitive. That is, if H and N are subgroups of G such that: $H \lhd N \lhd G$, it is not always the case that $H \lhd G$. When a user encounters this problem, he can however use a stronger property to retrieve the normality result, by showing that the smaller subgroup is characteristic.[2] Indeed:

$$\text{if } K \text{ char } H \text{ and } H \lhd G, \text{then } K \lhd G$$

IT IS THUS INTERESTING to make characteristicity proofs as easy as possible in the SSReflect libraries, and there is a common case of subgroups where a significant improvement is possible. The SSReflect libraries contain frequent references to subgroups defined for each group G. Apart from the Puig subgroup and the Thompson J subgroup — which are tightly linked to the proof of the Feit-Thompson theorem — those subgroups in fact appear in most group algebra textbooks. Browsing the literature, we noticed that the proof that those subgroups are characteristic is in fact treated two possible ways:

- Either the lemma is systematically left for the reader to prove at the beginning of a bundle of exercises, signaling a simple but somewhat tedious proof. This is the systematic habit of Gorenstein (2007), for example.

- Or, more rarely, the proofs are explicitly carried, but require a moderate amount of justification (around 6 lines) even for the simplest cases (Dixon 1973; Kurzweil and Stellmacher 2004; Rotman 1995, resp. pp. 78, 17, 104). For the COQ user, this naturally means that however how simple the reasoning involved is, he will have to reproduce a proof of approximately the same length.

Our objective for the group library is to make those mathematically simple but tedious developments quasi-instantaneous, requiring minimal effort on the part of the user. Table 3.1 on the next page lists all sixteen of the subgroups we deal with.

While some of those subgroups are specifically tailored for the proof of the Feit-Thompson theorem, most are of common use in group theory. It happens that they are all characteristic subgroups, but common literature does not use any shortcut to justify that: characteristicity proofs are either custom-made or, more often, left as exercises.

[1] **Definition 9.** *A subgroup* N *of* G *is* normal *if for all* $m \in N$ *and* $g \in G$, $gmg^{-1} \in N$. *That is, a subgroup is normal if it is invariant by conjugation by an element of the parent group.*

[2] **Definition 10.** *A subgroup* H *of* G *is* characteristic *if for every automorphism* φ *of* G, $\varphi(H) = H$. *We note this as* H char G.

Symbol	Name	Definition	SSReflect name
G	identity	G itself	`id`
1	trivial subgroup	the trivial subgroup $\{1\}$	`triv`
Z(G)	center	$\{z \in G \mid \forall g \in G, zg = gz\}$	`center`
$Z_n(G)$	the upper central series	(Gorenstein 2007, p. 21)	`ucn`
G'	derived subgroup	$< [g,h]_{g,h \in G} >$	`der`
$G^{(n)}$	the lower central series	$\underbrace{l(l(\ldots l}_{n \text{ times}}(G)\ldots))$, where $l(H) = H'$	`lcn`
$L_G(G)$	Puig subgroup	(Bender and Glauberman 1995, p. 139)	`Puig`
L(G)		$L(G) = \bigcap_{n \geq 0} L_{2n+1}(G)$, where $L_n(G) = L_G(L_{n-1}(G))$	`L_`
$L_*(G)$		$L_*(G) = \bigcup_{n \geq 0} L_{2n}(G)$, where $L_n(G) = L_G(L_{n-1}(G))$	`L_*`
$\Phi(G)$	Frattini subgroup	the intersection of all maximal subgroups of G	
F(G)	Fitting subgroups	subgroup generated by all nilpotent normal subgroups	`Fitting`
$O_\pi(G)$	π-core	the largest normal π-subgroup of G	`pcore`
$O_{\pi'}(G)$	π'-core	the largest normal π' subgroup of G	`pcore`
$O_{\pi,\pi'\ldots}(G)$	The upper π-series of G	(Gorenstein 2007, p. 226)	`pseries`
$O_{\pi',\pi,\ldots}(G)$	The lower π-series of G	(Gorenstein 2007, p. 226)	
$\Omega_i(G)$	Omega subgroup	the subgroup of the p-group G generated by its elements of order dividing p^i	`Ohm`
$\mho_i(G)$		the subgroup of the p-group G generated by the elements $(x^p)^i$ with x in G.	`Mho`

Table 3.1: The sixteen subgroups defined in the SSReflect library. Contrary to the literature, we chose to note the commutator (derived) subgroup with a lower case l, to ensure compatibility with the Puig functor. We remind that the π-subgroups (or Hall π-subgroups) involved *e.g.* in the `pcore` definition are those whose orders involve only primes in π and whose indices involve no primes in π. We include succint definitions when size permits.

The definitions of those subgroups have notable particularities:

- they include no assumption on the parent group G. In that sense, they are really *subgroup-defining functions* , defined on the class of all groups.
- they often stem from compositions of other subgroups: the definition of the lower central series uses that of the commutator subgroup, the definition of the π-series uses the π-core, etc

The first characteristic has repercussions on the type of the definition of those subgroup-defining functions in Coq, a matter we will study in chapter 3.4. The second quality, on the other hand, suggests to study how the characteristicity property (definition 10 on the preceding page) translates across the composition of some such subgroups: for example, given the high complexity of the definition of the upper central series, we can expect its characteristicity proof to be of the less enjoyable variety, unless we can provide good compositionality properties for its component subgroups.

Let us remember then the objects of our interest:

Definition 11 (subgroup-defining function)**.** *A* subgroup-defining function *is a mapping which to* any *group* G *associates a subgroup* H *of* G. *We will note subgroup-defining functions with capital letters.*

3.1.2 Subgroup-defining functions, invariance, and composition

Until § 3.4 on p. 131, we will denote the application of a set-lifted *group morphism* with prefix superscript (exponential) notation, in order to avoid any confusion with that of a subgroup-defining function.

Invariance properties The invariance properties we can expect from our sixteen subgroups can be stated in a more uniform manner. To begin with, since φ is injective and H is a finite subgroup of G, we can reformulate the characteristicity property in the following manner:

$$\mathrm{H}\ \mathsf{char}\ \mathrm{G} \Leftrightarrow \forall \varphi \in \mathsf{Aut}(\mathrm{G}), \mathrm{H}^{\varphi} \subseteq \mathrm{H} \tag{3.1}$$

Let F be a subgroup-defining function, that is a mapping from groups to groups, such that $\mathrm{F(G)} \leq \mathrm{G}$ for all G (here $\leq$ denotes the subgroup relation, whereas $\subseteq$ stands for set inclusion). Since for an automorphism $\mathrm{G}^{\varphi} = \mathrm{G}$, equation 3.1 becomes:

$$\mathrm{F(G)}\ \mathsf{char}\ \mathrm{G} \Leftrightarrow \forall \varphi \in \mathsf{Aut}(\mathrm{G}), \mathrm{F(G)}^{\varphi} \subseteq \mathrm{F}(\mathrm{G}^{\varphi}) \tag{3.2}$$

In fact, we have found it sufficient, and more compositional to study this property for a restriction of subgroup-defining functions to nearly-invariant subgroups.[3] Indeed, it allows us to study properties of objects which give rise in an unambiguous sense to a functor, in a sense we will explain and use but in § 3.4 on p. 131. For this reason, we call these particular subgroup-defining functions *functorials*:

[3] We remind that a subgroup H of G is *fully* invariant if for all φ isomorphism of G, $\mathrm{H}^{\varphi} = \mathrm{H}$. Note automorphisms are, by definition, *bijective endomorphisms*.

Definition 12. *Let* F *be a subgroup-defining function, then* F *is a* group functorial *if and only if:*

For any isomorphism φ *of domain* $\mathrm{G}, \mathrm{F(G)}^{\varphi} \subseteq \mathrm{F}(\mathrm{G}^{\varphi})$.

When this introduces no ambiguity, we will omit the "group" denomination.

As it turns out, most of the sixteen subgroup-defining functions mentioned above turn out to verify this equation for much more than automorphisms of their parent group. This prompts us to give the following two definitions:

Definition 13. *Let* F *be a subgroup-defining function, then* F *is a* strong group functorial *if and only if:*

For any morphism φ *of domain* $\mathrm{G}, \mathrm{F(G)}^{\varphi} \subseteq \mathrm{F}(\mathrm{G}^{\varphi})$

Definition 14. *Let* F *be a subgroup-defining function, then* F *is a* hereditary functorial *if and only if:*

For any morphism φ *of domain* $\mathrm{D}, (\mathrm{F(G)} \cap \mathrm{D})^{\varphi} \subseteq \mathrm{F}((\mathrm{G} \cap \mathrm{D})^{\varphi})$

In an equivalent, but less category-theoretic vocabulary, we say that a subgroup-defining function F verifying for all morphisms φ of domain G,

$$\mathrm{F(G)}^{\varphi} \subseteq \mathrm{F}(\mathrm{G}^{\varphi})$$

[4] Notice here that our definition does not involve the *codomain* or *range* of φ. An alternative possibility would have been to consider *continuous* those subgroup-defining functions such that $\forall$ morphisms $\varphi : \mathrm{G} \to \mathrm{H}, \mathrm{F(G)}^{\varphi} \subseteq \mathrm{F(H)}$. In that case, the center subgroup, for example, would not define a strong functorial. However, since our main concern was characteristicity proofs for groups, and since our definition of morphisms in Coq is range-free — which amounts to restricting morphism ranges to obtain their canonical surjection — this formulation seemed sufficient (see § 2.3 on p. 106).

is *continuous*,[4] and a subgroup-defining function F verifying for all morphisms φ of domain D,

$$(F(G) \cap D)^{\varphi} \subseteq F((G \cap D)^{\varphi})$$

is *hereditary*. It is clear that hereditary implies continuous. The term hereditary will seem more natural once remarked that:

Lemma 3. *Let* F *be a strong functorial (a.k.a continuous subgroup-defining function). Then* F *is a hereditary functorial (hereditary subgroup-defining function) if and only if* $\forall H \leq G$, $F(G) \cap H \subseteq F(H)$.

Composition operators Let us now come back to the way some of the subgroups mentioned in table 3.1 on p. 120 come up as composition of other such functorials. The lower central series, L(G) and $L_*(\bullet)$ are defined in terms of iterations of the following product:

Definition 15. *Let* F_1 *and* F_2 *be two functorials.. Then we define* $F_1 \nabla F_2$ *as the functorial that associates to any group* G *its image* $F_2(F_1(G))$, *and we call it the* lower product *of* F_1 *and* F_2.

On the other hand, the upper central series and the upper and lower π-series are iterations of the following product:

Definition 16. *Let* F_1 *and* F_2 *be two functorials. Then we define* $F_1 \triangle F_2$ *as the functorial that associates to any group* G *the inverse image of* $F_2(G/F_1(G))$ *by the quotient morphism induced by* $F_1(G)$, *and we call it the* upper product *of* F_1 *and* F_2.

$$Z_n(G) = \begin{cases} 1 & \text{if } n = 0 \\ Z(G) & \text{if } n = 1 \\ (\bullet \mapsto \bullet/Z_{n-1}(G))^{-1}(Z(G/Z_{n-1}(G))) & \text{otherwise} \end{cases}$$

For which we notice:

$$\forall n > 1, Z_n = Z \triangle Z_{n-1}$$

Figure 3.1: Example : the upper central series

Of our sixteen subgroup-defining functions, six are obtained by composing groups that are already the image of some functorial. Hence, **their characteristicity proofs become trivial once equipped with the appropriate composition properties**.

$$\begin{array}{ll}
\forall n > 1 & L_n(G) = L \nabla L_{n-1} \\
\forall n > 1 & Z_n(G) = Z \triangle Z_{n-1} \\
 & O_{\pi,\pi',\ldots} = \text{fixpoint}(G \mapsto O_{\pi} \triangle O_{\pi'}(G)) \\
 & O_{\pi',\pi,\ldots} = \text{fixpoint}(G \mapsto O_{\pi'} \triangle O_{\pi}(G)) \\
L(G) = \bigcap_{n \geq 0} L_{2n+1}(G) & L_n(G) = L_G \nabla L_{n-1} \\
L_*(G) = \bigcup_{n \geq 0} L_{2n}(G) & L_n(G) = L_G \nabla L_{n-1}
\end{array}$$

We have proved the following:

Theorem 5. *If* F *is a strong functorial, and* hF *is a hereditary functorial, then* $hF \triangle F$ *is a strong functorial. Moreover, if* F *is hereditary , then* $hF \triangle F$ *is hereditary too.*

The equivalent theorem for the lower product introduces the notion of *monotonic* subgroup-defining function:

Definition 17. *Let* F *be a subgroup-defining function. It is monotonic if and only if for all* G_1, G_2 *groups such that* $G_1 \subseteq G_2$, $F(G_1) \subseteq F(G_2)$.

In that case:

Theorem 6. *If* F *is a functorial, and* cF *is a monotonic functorial, then so does* cF ∇F. *Additionally:*
- *if* F *and* cF *are strong functorials, then so is* cF ∇F.
- *if* F *is monotonic, then so is* cF ∇F.

These two theorems are enough to let us build the characteristicity proofs of the appropriate subgroup-defining functions.

Table 3.2 lists the building blocks of our composition properties, that is the continuous subgroup-defining functions that we have proved to be *monotonic* or *hereditary*. Notice two exceptions:
- the Puig subgroup is a monotonous subgroup-defining function that verifies equation 3.1 on p. 121, but is not continuous.
- the Frattini subgroup is neither monotonic nor hereditary, but it is normal-monotonic.[5]

Monotonic	Hereditary
1, G, l(G), Ω_i(G), $\mho_i$(G)	1, G, Z(G), F(G), O_π(G)

Table 3.2: Basic subgroup-defining functions (not obtained through composition), arranged by composition class

[5] We have proven
Lemma 4. *For all* N, G *such that* $N \lhd G$, $\Phi(N) \subseteq \Phi(G)$.

WE CAN NOW CONCLUDE on our compound subgroup-defining functions. We have formalized the following results by composition:
- The upper and lower π-series, obtained from upper products involving the π-core, are hereditary subgroup-defining functions.
- The upper central series, obtained from upper products involving the center, is an hereditary subgroup-defining function.
- The lower central series, obtained from lower products involving the derived subgroup, is a monotonic subgroup-defining function.
- $\bullet \mapsto L(\bullet)$ and $\bullet \mapsto L_*(G)$, obtained from lower products involving the Puig subgroup, are monotonic subgroup-defining functions.

3.2 Radicals: from subgroup-defining functions to torsion theories

We do not introduce categorical vocabulary properly yet, since we will come back in a more detailed fashion to the specific literature piece alluded to here, in § 3.4.1 on p. 131.

To the best of our knowledge, functorials as such were first noticed by Eilenberg and MacLane (1945),[6] in the context of foundational work on category theory. The authors noticed that some generic definitions of subgroups benefited from invariance properties with respect to specific classes of group morphisms, and they showed $\mathfrak{Grp}$ was an example where looking at definition of mappings between objects was to be an integral part of their definition. The history of this attempt of classifying group-theoretic concepts in terms of invariance, and that of its impact, find a detailed treatment in Marquis (2006). However, the invariance thread quickly bifurcates away from groups, since the authors quickly found in a *topos* a more suitable construction, less dependent on the notion of subsets.

[6] With (Eilenberg and MacLane 1942) as an early interest.

However, subgroup-defining functions have come up in other, related branches of mathematics. Indeed, when investigating group-theoretical properties (like commutativity, or the property of being noetherian), one soon makes the following observations: (i) firstly it is possible to derive from these properties certain characteristic subgroups, and (ii) secondly these characteristic subgroups are a central part of the study of these properties.

We have been relatively conservative in our description of the literature on radicals and their application in non-abelian environments. For more details on this section, we point the reader to the valuable surveys of Márki (2009) and Gardner (2010). When they exist, we have always tried to give bibliographic references in English. Naturally, the initial publication date should not be confused with that of the translation.

For instance, the property of being *abelian* (i.e. a *commutative group*) "gives rise" to the commutator subgroup $[G,G]$ of a group G[7] (in a sense we are about to make precise):

- The commutator is the intersection of all co-abelian subgroups, that is, the subgroups $H \lhd G$ such that G/H is an abelian group.
- A group G is *abelian* if and only if $[G,G] = 1$.

[7] Also commonly noted G' (notably in the above), this subgroup is the subgroup generated by the *commutators*, the images of the map:

$$x, y \mapsto xyx^{-1}y^{-1}$$

for all $x, y \in G$.

We would then like to single out the class under consideration in such a way that the derived characteristic subgroups are in some sense amenable to treatment.

To make this general statement more precise, we will first consider that a group-theoretic property defines a *class* of groups, and talk of a *class* as of the associated group-theoretic property, henceforth. We let upper-case Zapf script letters (,…,) range over classes.

We start with a way to obtain functorials from a class of groups:

Definition 18. *Let $\mathcal{E}$ be a class of groups, and* G *any group.*

- $\mathcal{E}'(G)$ *is the join*[8] *of all normal $\mathcal{E}$-subgroups of* G.
- $\mathcal{E}^*(G)$ *is the intersection of all normal co-$\mathcal{E}$-subgroups, that is, the intersection of all* $N \lhd G$ *with* G/N *an $\mathcal{E}$-group (equivalently, the smallest normal subgroup of* G *such that the group factored by it is in $\mathcal{E}$).*

[8] Henceforth by *join* of groups A and B, we mean the group generated by their union $< A \cup B >$. Since this operation can be easily seen to be associative, its set-lifting is clear.

"Symmetry is a complexity-reducing concept ; seek it everywhere." (Perlis 1982)

We have similar definitions for functorials:

Definition 19. *Let us consider a functorial* F.

- *We say that* F *is* lower idempotent, *if for any group* G, $F(F(G)) = F(G)$, *or otherwise said* $F \nabla F = F$.
- *We say that* F *is* upper idempotent *if for any group* G, $F(G/F(G)) = 1$ *or otherwise said,* $F \triangle F = F$.
- *If a functorial* F *is* lower idempotent *and* upper idempotent, *we say it is* fully idempotent.

For example, the *abelianization* of a group, i.e. the subgroup $G/[G,G]$ is an abelian group. Since we remember the commutator of abelian groups is trivial, the commutator is *upper idempotent*.

Similarly to Definition 18, we define:

Definition 20. *Let* F *be a functorial, then:*
- F' *is the class of groups* G *such that* $F(G) = G$
- F^* *is the class of all groups such that* $F(G) = 1$

Then we can try to define, for properties, an analogous of strong functoriality for functorials:

Theorem 7 (Residual property)**.** *Let be a* group-theoretic *class (that is, implicitly closed by isomorphism) inherited by surjective images. In our "range-free" context, this means* $\forall G \in \mathcal{E}, G^{\varphi} \in \mathcal{E}$ *for* φ *any morphism (in the following, we will just say "images" for those* surjective *images). The following properties are equivalent:*

(i) a. Intersections of normal co-$\mathcal{E}$*-subgroup are co-*$\mathcal{E}$*-subgroups.*
b. Images of $\mathcal{E}$*-groups are* $\mathcal{E}$*-groups.*

(ii) a. $\mathcal{E}^*(G)$ *is for every* G *a co-*$\mathcal{E}$*-subgroup of* G*.*
b. Images of $\mathcal{E}$*-groups are* $\mathcal{E}$*-groups.*

(iii) The normal subgroup N *of* G *is a co-*$\mathcal{E}$*-subgroup if and only if* $\mathcal{E}^*(G) \subseteq N$

(iv) a. G *is an* $\mathcal{E}$*-group if and only if* $\mathcal{E}^*(G) = 1$
b. $\mathcal{E}^*$ *is a strong functorial.*

Any property verifying one of (i)-(iv) is called residual.

The same sort of construction also works to give, for properties, an analogous of the heredity property for functorials.

Theorem 8 (Co-residual property)**.** *Let be a* group-theoretic *class (again, implicitly closed by isomorphism) inherited by normal subgroups. The following properties are equivalent:*

(i) a. Products of normal $\mathcal{E}$*-subgroups are* $\mathcal{E}$*-groups.*
b. Normal subgroups of $\mathcal{E}$*-groups are* $\mathcal{E}$*-groups.*

(ii) a. $\mathcal{E}'(G)$ *is for every* G *an* $\mathcal{E}$*-subgroup of* G*.*
b. Normal subgroups of $\mathcal{E}$*-groups are* $\mathcal{E}$*-groups.*

(iii) The normal subgroup N *of* G *is a co-*$\mathcal{E}$*-subgroup if and only if* $N \subseteq \mathcal{E}'(G)$

(iv) a. G *is an* $\mathcal{E}$*-group if and only if* $\mathcal{E}'(G) = G$
b. $\mathcal{E}'$ *is an hereditary functorial.*

Any property verifying one of (i)-(iv) is called co-residual.

Proof : Theorem 7. (ii) ⇒ (i) is clear.

If (ii) is true, and $N \lhd G$ such that $\mathcal{E}^*(G) \subseteq N$, G/N is the image of $G/\mathcal{E}^*(G)$ by a canonical surjection and thus an $\mathcal{E} - subgroup$, making N a co-$\mathcal{E}$-subgroup. Since $\mathcal{E}^*(G)$ is by definition the intersection of all co-$\mathcal{E}$-subgroups, provided N is one such co-$\mathcal{E}$-subgroup, it is clear that $\mathcal{E}^*(G) \subseteq N$. Hence (ii) ⇒ (iii).

Assume (iii). If J is an intersection of co-$\mathcal{E}$-subgroups, then $\mathcal{E}^*(G) \subseteq J$, and J is a co-$\mathcal{E}$-subgroup. Moreover, if G is a -group, then 1 is a co-$\mathcal{E}$-subgroup, so that $\mathcal{E}^*(G) = 1$, which implies that $\mathcal{E}^*(G) \subseteq N$ for any $N \lhd G$. By the first isomorphism theorem, every image of G is an $\mathcal{E}$-group, hence (iii) ⇒ (i), and (i)-(ii)-(iii) are equivalent.

Assume (i)-(iii). As explained above, we have $\mathcal{E}^*(G) = 1$ if G is an $\mathcal{E}$-group, and (ii.a) proves that G is an $\mathcal{E}$-group if $\mathcal{E}^*(G) = 1$. We have (iv.a). Moreover, if φ is an morphism of domain G, then it induces a morphism from $G/\mathcal{E}^*(G)$ upon $G^{\varphi}/\mathcal{E}^*(G)^{\varphi}$, which is then an -group by (ii.b). Hence $\mathcal{E}(G^{\varphi}) \subseteq \mathcal{E}(G)^{\varphi}$ by (iii). The inverse image of the latter gives rise to an isomorphism $G/\mathcal{E}^*(G^{\varphi})^{\varphi^{-1}} \cong G^{\varphi}/\mathcal{E}^*(G^{\varphi})$. The latter is an $\mathcal{E} - group$ by (ii.a), so that it is also the case of the former and (iii) gives us:

$$\mathcal{E}^*(G)^{\varphi} \subseteq [\mathcal{E}^*(G^{\varphi})^{\varphi^{-1}}]^{\varphi} = \mathcal{E}^*(G^{\varphi}) \subseteq \mathcal{E}^*(G)^{\varphi}$$

We have (iv.b).

Assume (iv). Note φ the canonical surjection onto $G/\mathcal{E}^*(G)$, then (iv.b) gives us that

$$\mathcal{E}^*(G/\mathcal{E}^*(G)) = \mathcal{E}^*(G^{\varphi}) = \mathcal{E}^*(G)^{\varphi} = \mathcal{E}^*(G)/\mathcal{E}^*(G) = 1$$

Hence, $G/\mathcal{E}^*(G)$ is an $\mathcal{E}$-group, and (ii.a) is true. If moreover σ is a morphism, and since $\mathcal{E}^*(G) = 1$ from (iv.a), we have $\mathcal{E}^*(G^{\sigma}) = 1$ by (iv.b). We can then conclude using (iv.a) that G^{σ} is an $\mathcal{E}$-group, proving (ii.b) and the equivalence of (i)-(iv). □

Proof: Theorem 8. (ii) ⇒ (i) is clear.

Assume (ii), then if $N \lhd G$, $N \lhd \mathcal{E}'(G)$ by definition of the latter, and (ii.a), (ii.b) prove the equivalence (iii).

Assume (iii). Applying it to $\mathcal{E}'(G)$ itself tells us that it is an $\mathcal{E}$-group, and, knowing that $\mathcal{E}'(G) \subseteq G$, it implies (iv.a). Moreover, if $N \lhd G$ then $\mathcal{E}'(N)$ is a characteristic subgroup of N, and, as mentioned in § 3.1.1 on p. 119, a normal subgroup of G. Hence $\mathcal{E}'(N) \subseteq \mathcal{E}'(G)$, so that $\mathcal{E}'(N) \subseteq N \cap \mathcal{E}'(G)$. Finally, $N \cap \mathcal{E}'(G)$ is a normal subgroup of $\mathcal{E}'(G)$ and as such (iii) proves it is an $\mathcal{E}$-group. We can use this to apply (iii) again considering $N \cap \mathcal{E}'(G) \lhd N$, so that $N \cap \mathcal{E}'(G) \subseteq \mathcal{E}'(N)$. Thus (iv.b) is valid.

Assume (iv). If G is a product of normal $\mathcal{E} - subgroups$, then $G = \mathcal{E}'(G)$ by definition of $\mathcal{E}'$, so that (iv.a) lets us conclude to (i.a). Moreover, if $N \lhd G$ and G is an $\mathcal{E}$-group, then (iv.b) implies

$$\mathcal{E}'(N) = N \cap \mathcal{E}'(G) = N \cap G = N$$

, and that we can conclude that N is also an $\mathcal{E}$-group from (iv.a). This lets us conclude the proof of (i) and therefore (i)-(iv) are equivalent. □

Then we have the following theorem:

Theorem 9.

(i) The functorial F *is hereditary, if and only if* F' *is a co-residual class and* $F'' = F$,

(ii) The functorial F *is strong, if and only if* F^* *is a residual class and* $F^{**} = F$,

(iii) The class $\mathcal{E}$ *is co-residual, if and only if* $\mathcal{E}'$ *is hereditary and* $\mathcal{E}'' = \mathcal{E}$,

(iv) The class $\mathcal{E}$ *is residual, if and only if* $\mathcal{E}^*$ *is strong and* $\mathcal{E}^{**} = \mathcal{E}$.

In sum, those lemmas create a link between, on the one hand, the properties of the join (or intersection in the case of co-radicality) of a selective class of groups that singles out a given property, and, on the other hand, the value of some subgroup-defining functions (equipped with some properties).

For instance, the functorial F is of the form $F = \mathcal{E}'$ for a co-residual property $\mathcal{E}$ such that normal subgroups are $\mathcal{E}$-groups if and only if:

$$F(N) = N \cap F(G) \text{ for every } N \lhd G$$

Note that we approach those notions with a special focus aimed at leveraging properties of subgroup-defining functions- hence, we do and will insist on the right side of that link. It is contrary to the most common approach : in most of the literature, it is that join of elements of a class of groups that is mostly studied, and from which “a radical” is defined.[9]

Nonetheless, the study of those objects was founded during the 1950s by Amitsur and Kuroš (Amitsur 1952, 1954a,b; Kuroš 1973a), as a generalization of the concept of largest nilpotent ideal of a finite dimensional associative algebra.

The generalization endeavour quickly turned towards an abelian setting, in which the normality of subgroups (such as the one encountered in the pre-radicality condition above) is obtained without further ado. This also offers the advantage of being in a framework where it is always possible to consider the direct sum above instead of the join. In that context, *radical* theory is thus simply the study of the direct sum of all subgroups in a class of groups.

Radical theory is by now a part of ring theory, giving its name to the famous instance that is Jacobson’s radical.[10] Nonetheless, we can notice that of our 16 subgroup-defining functions, a good number are defined by intersection of subgroups, or join of quotient groups, hinting at some relevance of radical objects in our case. What is more, the parenthood with group theory, and the subgroup-defining functions of particular interest in this thesis can be traced precisely in radical theory: it is of note that Amitsur explicitly makes a reference to MacLane’s foundational work (MacLane 1950) stemming from group theory, and based on the exact study of those subgroup-defining functions to define bicategories[11] Finally, radical theory for groups was developed explicitly (Gardner 1989; Kuroš 1973b), if sporadically, and some decades after the analogous theory for rings. General radical theory for groups is therefore somewhat relevant to our interests. It has known a recent relative renewal (Gardner 2010) which, however, does not focus on subgroup-defining functions.

We are less ambitious than wanting to develop an exact correspondence

Proof: Theorem 9. We only prove the difficult case, i.e. Showing that we have co-residuality (resp residuality) in (i),(ii), assuming the heredity of F (resp. that F is strong).

Assume F is strong. We will prove case (iii) of theorem 7. If G is an F^*-group and φ a morphism, then $F(G^\varphi) = F(G)^\varphi = 1$ since F is strong and by definition of F^* applied to G. G^φ is then an F^*-group, and images of F^*-groups are F^*-groups.

Let now σ denote the canonical isomorphism upon $G/F(G)$. Then $F(G/F(G)) = F(G)^\sigma = 1$ so that $G/F(G)$ is an F^*-group. Then if $N \lhd G$ G/N is the image of F^*-group $G/F(G)$, and thus an F^*-group. Thus by strength of F and the definition of F^*:

$$1 = F(G/N) = N.F(G)/N$$

which implies $F(G) \subseteq N$. N is therefore a co-F^*-subgroup of G if and only if $F(G) \subseteq N$, i.e. (iii) is valid.

Assume now that F is hereditary. We will prove case (i) of theorem 8. If $N \lhd G$ with N a F'-subgroup of G, then

$$N = F(N) = N \cap F(G) \subseteq F(G)$$

In particular, if G is the product of normal F'-subgroups, $G = F(G)$ so that products of normal F'-subgroups are F'-subgroups. If now $N \lhd G$ is a subgroup of the F'-group G, then

$$F(N) = N \cap F(G) = N \cap G = N$$

Hence, normal subgroups of F'-groups are F'-groups. Hence F' is co-residual. □

[9] A *radical* in this sense is a class of rings $\mathcal{S}$ such that:

(1) the homomorphic image of a ring in $\mathcal{S}$ is also in $\mathcal{S}$.

(2) every ring R contains an ideal S(R) in $\mathcal{S}$ which contains every other ideal in $\mathcal{S}$

(3) $S(R/S(R)) = 0$. The ideal S(R) is called the radical, or $\mathcal{S}$-radical, of R. For instance, the *Jacobson radical* is the intersection of all primitive ideals of a ring, called the upper radical generated by the class of all primitive rings.

[10] If I is an ideal of the commutative ring R, Jac(I) is the intersection of all maximal ideals containing I.

between the most common expressions for a radical, and our subgroup-defining functions, and prefer to focus on the few publications close to our concern (Baer 1966; Plotkin 1969, 1983):

> Of course, abstract group-theoretical funtions in implicit form have been studied for a long time, but only in the recent paper by Baer (1966) have they been named explicitly and regarded as an object in their own right.
>
> (Plotkin 1983)

In general, the lemmas we developed are picked out of conflicting sets of definitions, in which we have taken the most general solution, and that we essentially adapted to cover the slightly different setting of our surjective representation of morphisms.

Indeed, where we define continuity as such:

$$\forall f, F(G)^f \subseteq F(G^f)$$

because our morphisms have no pre-defined range, a more mathematical approach would state

$$\forall f : G \to H, F(G)^f \subseteq F(H)$$

This second definition has composition properties that are more regular. For example, the preservation of the continuity across the composition of two functorials is automatic in the "mathematical" case.

Beyond this translation of properties of subgroup-defining functions, a small part of theories developed from radicals still interests us, namely torsion theories. Torsion theories develops relations between dual classes of groups, and attaches itself to closure of those classes of groups by operations such as taking a subgroup, a quotient group, a group extension, or an isomorphic image.

[11] *"The whole theory can be developed in a far wider class of mathematical objects. The larger field in which this can be done is that of the Lattice-ordered bicategories of MacLane defined in 1950, which satisfy some additional properties so that the two main isomorphisms hold, and some minor properties of ideals in rings. (...) It is worth noting that the whole theory of radicals is just a relation between injections and projections of a bicategory. We just use 'objects', 'normal subobjects', 'supermaps' instead of 'rings', 'ideals', and 'homomorphisms'."*

(Amitsur 1954b)

3.3 *Torsion theories for groups*

If some radical theory was already enough to make some link between group functorials and group-theoretic properties expressed as classes of groups, it was not enough to capture meaningful classes we wanted to work with.

3.3.1 *Definitions and Properties*

We start with a few preliminary notations:

Definition 21 (Null morphism)**.** *If $f : X \to Y$ is a morphism such that $f(x) = 1_Y$ for all $x \in X$, then we call f null and note it by $0_\to$.*

Given a class of groups, we can define *class constructors*:

Definition 22 (Torsion theory)**.** *Let $\mathfrak{T}$ and $\mathfrak{F}$ be two classes of groups. We define:*

$$\mathfrak{T}^r = \{H \in \mathfrak{Grp} \mid \forall G \in \mathfrak{T}, Hom(G,H) = 0_\to\} \tag{3.3}$$
$$\mathfrak{F}^l = \{G \in \mathfrak{Grp} \mid \forall H \in \mathfrak{F}, Hom(G,H) = 0_\to\} \tag{3.4}$$

$\mathfrak{T}, \mathfrak{F}$ is called a torsion pair *if the following holds:*

$$\mathfrak{T} = \mathfrak{F}^l \wedge \mathfrak{F} = \mathfrak{T}^r$$

Remark. *For any two classes $\mathfrak{T}, \mathfrak{F}$:*

$$\mathfrak{F} \subset \mathfrak{T}^r \Leftrightarrow \mathfrak{T} \subset \mathfrak{F}^l$$

We have proven:

Lemma 5. *Let F be a functorial, we define:*

$$\mathfrak{T} = \{G \in \mathfrak{Grp} \mid F(G) = G\} \tag{3.5}$$
$$\mathfrak{F} = \{G \in \mathfrak{Grp} \mid F(G) = 1\} \tag{3.6}$$

then we have:

$$\begin{cases} \mathfrak{T} \subset \mathfrak{F}^l \wedge \mathfrak{F} \subset \mathfrak{T}^r & \text{if F is monotonous} \\ \mathfrak{F}^l \subset \mathfrak{T} & \text{if F is upper idempotent} \\ \mathfrak{T}^r \subset \mathfrak{F} & \text{if F is lower idempotent} \end{cases}$$

Lemma 6. *Let $\mathfrak{T}, \mathfrak{F}$ be two group classes. Then, if $\mathfrak{T} = \mathfrak{F}^l$, $\mathfrak{T}$ is closed by:*

- *group isomorphism*
- *quotient group*
- *group extension*
- *group* join *(see note 8 on p. 124)*
- *direct sum*

Lemma 7. *Let $\mathfrak{T}, \mathfrak{F}$ be two group classes. Then, if $\mathfrak{F} = \mathfrak{T}^r$, $\mathfrak{F}$ is closed by:*

- *group isomorphism*
- *subgroup*
- *group extension*
- *(external) direct product*

Lemma 8. *Let $\mathcal{T}$ be a group class, and pose $\mathcal{F} = \mathcal{T}^r$. Then if $\mathcal{T}$ is closed by group isomorphism, quotient group, group extension, and join, $\mathcal{F}'$ is a fully idempotent radical. Moreover,*

$$\mathcal{T} = \{G \in \mathfrak{Grp} \mid F'(G) = G\}$$

$$\mathcal{F} = \{G \in \mathfrak{Grp} \mid F'(G) = 1\}$$

Lemma 9. *Let $\mathcal{F}$ be a group class, and pose $\mathcal{T} = \mathcal{F}^l$. Then if $\mathcal{F}$ is closed by group isomorphism, subgroup, group extension, and direct product, $\mathcal{F}'$ is a fully idempotent radical. Moreover,*

$$\mathcal{T} = \{G \in \mathfrak{Grp} \mid F'(G) = G\}$$

$$\mathcal{F} = \{G \in \mathfrak{Grp} \mid F'(G) = 1\}$$

3.3.2 Coq Formalization

The formalization of those results was relatively straightforward, and allowed us, for example, to connect the derived subgroup with the class of solvable groups, a torsion-free class of groups. Here is our definition of group class:

```
Definition trivlsed gT (B : {group gT}) hT (C : {group hT}) :=
  ∀f : {morphism B ↦ hT},
  f \ensuremath{\itbox{* B ⊆ C → f }}* B :=: 1%G.

Definition GClass := ∀gT (B: {group gT}), Prop.

Identity Coercion fun_of_obsel : GClass ↦ Funclass.

Implicit Types P Pa Pb : GClass.
```

We note the necessary polymorphism of classes of groups, caused by the fact that some of them must be invariant by taking the quotient (and since our quotient types are distinct from the original, unquotiented one). The inclusion relations between classes are the following:

```
Definition lP Pa Pb := ∀gT B, Pa gT B → Pb _ B.
Definition eP Pa Pb := ∀gT B, Pa gT B ⟷ Pb _ B.
```

With this definition, they allow a group with a given `finGroupType` and its quotient to be in the same class. This genericity has a downside: in the final results on the creation of a functorial from an intersection of groups we were required, however, to fix the `finGroupType` of the group we would work with. We had to make sure that the monomorphic intersection of groups we created was meaningful, something we chose to represent with a reflection of the membership predicate to booleans:

```
Variables Bbool Cbool : ∀gT, pred {group gT}.
Hypothesis reflectB : ∀gT (G:{group gT}), reflect (Bs G) (Bbool
    G).
```

This implies our interpretation of class `Bs` in terms of a functor will only be valid when `Bs` is not empty. We will have to carry around non-emptyness hypothesis, but this is actually better than the alternative, which is to have a `T` that does not always carry a group structure (being sometimes empty) - hence one that wouldn't even fit our functor structure.

3.3.3 Related Work

Torsion theory was developed for rings by Dickson (1966), but to our knowledge, there is no clear port of this theory to groups. Some precursors exist in unpublished form. (Ohtake 2008) defines a torsion theory for groups, but without concern for operations other than subgroups and group extensions, and with an exclusive focus on torsion-free theories. In a different take that aims at exploring the consequences of closure operators for groups, López Gerena (2009) develops some torsion theory for group classes, but his equivalence between the torsion-free theory of a functorial and a class of groups with suitable invariance requires much more invariance hypotheses than ours. None make the link with residuality.

3.4 Relational Parametricity

ALONG THIS EQUIVALENCE BETWEEN GROUP FUNCTORIALS AND CLASSES OF GROUP EQUIPPED WITH INVARIANCE PROPERTIES, we have developed an insightful way to look at group properties. We now want to aim at an even more insightful way to look at group functorials.

Until now, we have only looked at mathematical properties of those functorials. We now want to look at characteristics of their type-theoretic representation. We firstly make a link between the notion of functoriality and the continuity property we isolated in § 3.2 on p. 124. We then recognize that this continuity property happens to be a "free theorem" : a corollary of a relational parametricity theorem. Provided we can make that intuition precise in COQ, this would provide us with a way of obtaining group-theoretical properties from a subgroup-defining function, simply by looking at its type. After having made the connection with relevant portions of the literature on parametricity, we turn toward obtaining a parametricity result within COQ. Indeed, the key difficulty on which we focus is not the notion of parametricity itself, but the fact that it is a meta-theorem of any calculus for which it is valid. Nonetheless, we work within COQ and, faithful to the SSReflect approach, we do not want to add any parametricity axioms to the calculus. Our dilemma is thus to obtain a result that we could exploit, which is why we focus on reflection. We explain the insight that led us on the track towards making parametricity exploitable within the calculus, and provide a proof-of-concept example realizing that suggestion.

3.4.1 From subgroup-defining functions to Functors

The definition of continuity presented in section 3.1.2 on p. 121 has a more general justification, first noticed by Eilenberg and MacLane:[12] the definitions of those subgroups, defined for any arbitrary parent group, can be seen as object mappings of subfunctors[13] of the identity functor on (at least) the core[14]category of $\mathfrak{Grp}$.

They prove:

Lemma 10. *(Eilenberg and MacLane 1945, §14.1) Let* T *be a covariant functor with values in the category* $\mathfrak{Grp}$*, while* T' *is a functions that assigns to each object* G *a subgroup* G' *of* $\mathrm{T}(\mathrm{G})$*. Then* T' *is the object mapping of a subfunctor of* T *if and only if for each* $f : \mathrm{G} \to \mathrm{H}$,

$$\mathrm{T}'(\mathrm{G}))^{\mathrm{T}(f)} \subseteq \mathrm{T}'(\mathrm{H}) \quad (3.7)$$

If T' *satisfies this condition, the corresponding mapping function* $\mathrm{T}'(f)$ *is uniquely determined as the restriction of* $\mathrm{T}(f)$ *to the domain* $\mathrm{T}'(\mathrm{G})$*.*

The characteristicity condition then becomes a direct corollary obtained by substituting an automorphism φ, the identity functor, and the image of G by the subgroup-defining function, for respectively f, T and $\mathrm{T}'(\mathrm{G})$ in previous equation 3.7.

The categorial vocabulary introduced here may seem unnecessarily complicated just to prove that some subgroups are characteristic. However, the

[12] S. Eilenberg and S. MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58(2):231–231, 1945. ISSN 0002-9947. doi:10.1090/S0002-9947-1945-0013131-6

[13]

Definition 23. *Let* $\mathrm{T}_1, \mathrm{T}_2 : \mathcal{C} \to \mathcal{D}$ *be two functors of same variance. The functor* T_1 *is said to be a* subfunctor *of* T_2 *if for all objects* $\mathrm{G} \in \mathcal{C}$, $\mathrm{T}_1(\mathrm{G}) \subset \mathrm{T}_2(\mathrm{G})$ *and* $\mathrm{T}_1(f) \subset \mathrm{T}_2(f)$ *for all* $f : \mathrm{G} \to \mathrm{H}$ *in* $\mathcal{C}$*.*

[14] The *core* of a category C is the subcategory consisting of all objects, but which takes only the bijective homomorphisms as morphisms.

category on which a subgroup-defining function can be seen as giving rise to a subfunctor is usually not limited to the core category of $\mathfrak{Grp}$. For instance, for any group morphism f of domain including the group G,

$$f(G') \subseteq (f(G))'$$

where the quotation mark denotes the *commutator subgroup*. Hence, the commutator subgroup is a subfunctor of the identity on $\mathfrak{Grp}$ as a whole: in fact, we even have $f(G') = (f(G))'$. On the other hand, the image of the center subgroup is not always included in the center of the image of its parent,[15] but this property becomes true once we restrict ourselves to images by *surjective* morphisms.

Hence, Eilenberg and MacLane(Eilenberg and MacLane 1945) go on to remark that :

> various types of subgroups of G may be classified in terms of the degree of invariance of the "subfunctors" of the identity which they generate.

[15] Let us consider the direct product $A \times C$ of an *abelian group* A with C a *centerless group* (a group with trivial center) such that C contains a subgroup B isomorphic to A ($B \cong A$). Let us call σ the isomorphism that sends A to B. It is clear that A is the center of $A \times C$. Now, consider the endomorphism

$$f(A \times C) \mapsto C$$
$$x \mapsto \sigma \circ \pi_1(x)$$

where π_1 is the projection from $A \times C$ to A. Then the image by f of the center A of $A \times C$ is B, but the center of the image of f, C, is 1 by construction.

To make this construction work, consider that a dihedral group D_n of order $2n$ is centerless if n is odd. Moreover, a dihedral group is made, by construction, of the product of a cyclic group by an involution, so that this cyclic group furnishes a suitable abelian group with an isomorphic image in the dihedral. Hence, in the above, we can take $C = D_3$ and $A = \mathbb{Z}/2\mathbb{Z}$.

3.4.2 *Many Notions of Parametricity*

The notion of parametricity (coined informally by Strachey (2000) in 1967), in a nutshell, it states that a term of polymorphic type preserves relations between types: if a term u of the second-order polymorphic λ-calculus has type $\forall \alpha$:`Type` . σ and $R \subset \tau \times \tau'$ is a relation, then

$$u(\tau)(\sigma[R])u(\tau')$$

where $\sigma[R]$ is a relational interpretation of the type σ defined inductively over the structure of σ. Equivalently, parametricity could be defined as the identity extension property: for all terms of type u, v of type $\sigma(\alpha)$,

$$u(\sigma[\overrightarrow{eq}_\alpha])v \Leftrightarrow u = v$$

It finds its formal origin in the isomorphism between the second-order polymorphic λ-calculus F_2 and the second order intuitionistic predicate logic Π_2. The fact that any function provably total in Π_2 can be represented by an F_2 term is Girard's *representation theorem*. The fact that for a suitable notion of logical relation, every term in F_2 takes related arguments into related results provides a reverse embedding of F_2 into Π_2.[16] It is Reynold's *abstraction theorem* (Reynolds 1983).

[16] P. Wadler. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1-3): 201–226, May 2007. ISSN 03043975. doi:10.1016/j.tcs.2006.12.042

From this theorem and its original correspondence, an impressive body of literature has emerged during the last few decades. The first type of literature concerns the justification and reformulation of the original relational model presented by Reynolds, whose proof of the abstraction theorem was originally based on a Kripke semantics. Many authors aimed to give it a more pure logical foundation, or a categorical semantics. Notably, Plotkin and Abadi (1993) developed a logic for parametric polymorphism — that is, a second order logic over system F terms extended with an axiom schema expressing relational parametricity. Of note in this strand of work is the work of Abadi et al. (1993) who manages to provide a syntaxical proof over system F extended with a delimited axiom rule inspired from the identity extension property above. It is the only proof which does not —- to our knowledge — base itself upon a denotational semantics.[17]

[17] The categorical approach to parametricity sadly hits the tender spot of a subject where there is too much to say (because the link between functoriality has been made early, starting at least with the work of Freyd on PER models in 1990), and where the subject is slightly too distant to inform our hands-on approach (though we will mention a late offspring of Joyal's work in § 3.4.5 on p. 136)

The second large trend of work born from the abstraction theorem consists in extending these results to calculi richer than System F. Indeed, it is unclear which form of the correspondence exposed above holds for calculi augmented with features such as higher-kinded types, not to mention coinductive types or existentials — in particular, the equivalence of the identity extension (or what its analogous should be) and some relational interpretation of parametricity is no longer clear. Working in the context of practical programming and in Coq, we are, of course, interested in but a limited number of those extensions. The curious reader will gain much from the survey of Birkedal and Mø gelberg (2004),[18] after which he will forgive us for concentrating exclusively on the strand of work which seems more promising for our interest : Vytiniotis and Weirich (2010) developed a proof of relational parametricity for system F_ω which led to a recent extension to dependent polymorphic PTS (under some conditions) by Bernardy et al. (2010). We are hopeful for a future extension of their results to type theory.

Most of those proofs focus decidedly on relational parametricity and are not shy of a denotational approach, if only because all practical approaches to parametricity theorem have been corralled and motivated by Wadler's exposition of the notion of parametricity.[19]

In this paper, Wadler reformulates Reynolds' results and shows that they can be used to reason with parametric functions on the simple basis of their type. In particular, he makes the link between the notions of data abstraction and polymorphic types effective, by showing that polymorphic functions cannot inspect the data of their arguments and that from this insight, equations between terms can be rigorously derived. For instance, he shows that for "container" types, such as lists:

If `(f : ∀A. list A → list A)` and `g : T → U`, then

$$\texttt{map g} \circ \texttt{f}_\texttt{T} = \texttt{f}_\texttt{U} \circ \texttt{map g}$$

The practical approach to deriving those equations from polymorphic types and using them in polymorphic languages (mostly Haskell) form the third notable strand of work on parametricity. Since we are interested in the study of concrete subgroup defining functions, our investigation is naturally of that persuasion.

To make this interest clearer we now give an exposition of the core reasoning expressed in (Wadler 1989).

3.4.3 *Parametricity and relations*

Let us consider a typed relation α on λ-terms, which, for the sake of argument, we'll consider binary. Say that relation α relates λ-terms of types A and A′, for its first and second components.

Let us define functions on relations.

$\Phi = (\varphi, \varphi')$ is a function from relations to relations if for all $(x,y) \in \alpha$, $(\varphi(x), \varphi'(y)) \in \Phi(\alpha)$. If $\varphi : A \to B$ and $\varphi' : A' \to B'$, then $\Phi(\alpha)$ relates B to B′.

To avoid any confusion, we will call those functions *relation transformers*. Now, since our relations are typed, a definition of polymorphic relation transformers emerges naturally:

[18] If he is interested in type theory, he might also want to glimpse at the early unpublished paper of Takeuti (2001), which is the last scion of the aforementioned "logical" approach to launch a tendril in the direction of dependent types.

[19] P. Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89*, number June, pages 347–359, New York, USA, 1989. ACM Press. ISBN 0897913280. doi:10.1145/99370.99404

Ψ is a polymorphic relation transformer of type $\forall\alpha, \Phi(\alpha)$ if and only if for any relation β relating types B, B', then $\Phi: B \to \Phi(\beta)$ is a relation transformer.

What relational parametricity tells us is that polymorphic functions give rise to polymorphic relation transformers of the same type. More precisely:

If f is a polymorphic λ-function of polymorphic type, say

$$\forall A, [A] \to [A]$$

then (f, f) is a polymorphic relation transformer of type

$$\forall\alpha, [\alpha] \to [\alpha]$$

i.e. $(f\,B, f\,C)$ is a monomorphic relation transformer of type $[B] \to [C]$ for all α relating B to C.

The list, here represented by brackets, is just here to make things not-too-trivial, and to show that we extend relations to constructors. Let us see an example with rev, the polymorphic function that mirrors lists.

What *Theorems for free* tells us is that it might be useful to look at functions seen as relations. The typed binary relation $\theta(f)$ on λ-terms induced by a monomorphic λ-function $f: B \to C$ is evidently the set of pairs:

$$\theta(f) = \{(x:B, y:C) \mid y = f(x)\}$$

Now, we can easily build an extension of that relation to lists, as given by the map function:

$$[\theta(f)] = \{(l_1:[B], l_2:[C]) \mid l_2 = (\mathrm{map}\, f)\, l_1\}$$

Since (rev,rev) is a polymorphic relation transformer, (rev B, rev C) is a relation transformer. This just means that the transformation on relations given rise to by rev is just as closed as what the type of the original polymorphic function — returning lists of the same type as its argument — was. More precisely, for all $(l_1:[B], l_2:[C]) \in [\theta(f)]$ for some f, we have that $(\mathsf{rev}\,B\,l_1, \mathsf{rev}\,C\,l_2) \in [\theta(f)]$.

We now unfold the definition of $[\theta(f)]$ to notice that:

$$\mathsf{rev}\,C\,l_2 = (\mathrm{map}\, f)(\mathsf{rev}\,B\,l_1)$$

Recalling the definition of $[\theta(f)]$ for the original l_1, l_2 yields:

$$l_2 = (\mathrm{map}\, f)\, l_1$$

Which replaced in the above allows us to conclude:

$$\mathsf{rev}\,C(\mathrm{map}\, f)\, l_1 = (\mathrm{map}\, f)(\mathsf{rev}\,B\,l_1)$$

We have proven that rev and map f commute, as was our goal all along.

3.4.4 *Functoriality properties and subgroup-defining functions*

PROOF BY REFLECTION is by now a well-known staple of proof automation in COQ. The `ring` tactic (Grégoire and Mahboubi 2005) is the

posterchild for the efficiency of such an approach, and we have already mentioned it when explaining our own favor of reflection in § 2.1.1 on p. 84. However, the key difference between `ring` and small scale reflection is that the first approach occurs outside of the calculus, and can potentially reflect all the terms of COQ. On the other hand, small scale reflection is based on an inductive reflection predicate which limits its scope: it can only reflect terms using user-provided proofs of that predicate. Moreover, it is only so easy to use because it has had a little help from a top level tool: SSReflect. As such, its ambition is not to provide a way to reflect all the terms of COQ 's calculus. Thankfully, neither is ours.

At the time of this writing, no relational parametricity results have been proven for the whole calculus of COQ, and the theorem most of us are used to thinking of is still the one of System F. It is unclear that the cumulative calculus of inductive constructions with universes (§ 1.1.5 on p. 23) would support such a theorem. The situation gets even murkier if the calculus is extended with the whole set of features of COQ (including e.g. coinductive types).

However, reflection can be implemented within calculus as a decompilation function which recovers the syntax of a program, provided it belongs to a reasonable, well-chosen fragment of the extended PTS of COQ. Such a method has been implemented for the simply-typed λ--calculus in COQ,[20] but the definition of this reflection function involved a mutually-recursive, type-directed, dependently-typed decompilation function, and extending it to support a stronger calculus is still an open problem.

[20] F. Garillot and B. Werner. Simple Types in Type Theory: Deep and Shallow Encodings. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 368–382. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-74590-7. doi:10.1007/978-3-540-74591-4_27

Even though, our idea consists in working along those lines, in the sense that we are not ambitious for more than a relational parametricity theorem for the vanilla system F —- yet. However, we are ambitiously going for a way to connect live COQ terms with such a relational parametricity theorem, proven on a deep embedding of system F in COQ: we want to reify just those terms of COQ that correspond to valid system F terms. Once this is done, we hope that a relational parametricity result on the *reified* terms will be enough to obtain a free theorem on relevant COQ terms.[21]

[21] Granted, this may not suffice to prove the continuity property that we have encountered in § 3.2 on p. 124, since it involves no less than nested inductives, but we still hope it will turn out to be a step in the right direction

The method we have chosen to implement reflection is an application of the original insight that led us to develop a semantics for deterministic overlapping instances in § 1.4.4 on p. 74. Let us start by going back to what reflection is: On the one hand, we have a deep embedding of the calculus we want to reflect — that is an inductive type giving the BNF grammar of the syntax of said calculus. On the other hand, we have a shallow embedding of the calculus: select terms of COQ that can be seen as members of the calculus and that we want to reify. Interestingly, those COQ terms come with their type — that is they are a representation à la Church.

A full reflection procedure consists in giving a bijection between the two. That bijection can link equal terms in a certain sense: it can link one term of the deep embedding with the term that would be syntactically equal if written by a meta-theoretic observer. Another approach consists in linking terms that are equivalent modulo a given relation: for example, we can relate the normal form of a lambda-term of the deep embedding with the corresponding normal form of a COQ term of the shallow embedding, where "corresponding" designates the equality above — this was the approach of

(Garillot and Werner 2007). In any case, this consists in examining the structure of a term in any of the two embeddings and replicating it in the other one. In general, the difficult part of that correspondence is in writing the function that goes from the shallow embedding to the deep embedding: writing its inverse is easy for calculi that are sub-part of the one of COQ— it consists in writing an interpreter. In all cases, the procedures of that bijection are *defined recursively*: this of course, reflects that the structure of the calculus itself is inductive, as witnessed by the representation of its grammar using an inductive, in the deep embedding.

The key insight of reification using canonical structures is that, very often, observing the weak head normal form of a term is enough to select which case we are dealing with, to replicate the corresponding shallow construction, and to defer to a recursive call. Compositionality is the principle that allow us to implement said recursive call using `Canonical Structures`: in effect, the head constant of a normal term gives us enough information to recognize which kind of terms we are facing and we can then subvert the unification procedure to realize the recursive call for us — and build the sub-structures that will compose the rest of our term.

However it is difficult, simply using this machinery, to pass context to the recursive calls. The solution we need to implement is to pass a single, unchanging environment to the recursive calls — hence, we had to implement a search function for variables of the environment based on type declarations and unification. This is achieved by (Cohen 2010, annex C) using a CPS-style semantics based on the techniques we exposed in 1.4.4 on p. 74, and what we simply aim to extend to a bigger calculus.

3.4.5 Related work

Of the few concrete applicable tools for using parametricity while programming "in the wild", we must mention two outliers which inch closer than average towards our approach. Voigtländer has developed, along with a thorough analysis of what parametricity means in the context of Haskell (e.g. Voigtländer 2009), a tool to derive parametricity equations (Johann and Voigtländer 2004; Seidel and Voigtländer 2010).

Moreover, of the numerous category-theoretic approaches to parametricity, one can be developed — to a point — within a deep embedding in type theory, and more precisely in COQ. It therefore permits a unique approach to having some free theorems available as-is in the prover: the notion of *containers* has been developed by Abbott (Abbott 2003; Abbott et al. 2003) to give a functorial rendering of parametricity : containers are defined as types which access data through a containing layer which represents a uniform indexation of data through integers. The use of this layer makes some free theorems provable from a generic construction. The development in COQ of a part of this theory has been carried out by Prince et al. (2008). While stimulating, the construction is developed on the example of first-order polymorphic lists and will -- to the best of our understanding- require a significant improvement of proof-irrelevance in the calculus of COQ to be generalizable to arbitrary constructors. This innovative approach therefore clashes with the non-axiomatic stance of the Mathematical Components

team in SSReflect.

3.4.6 Future work

A developement of the extension of the reflection technology of (Cohen 2010, annex C) is ongoing. It will reify those terms of Coq that correspond to System F terms.

Conclusion

OUR WORK HAS THREE MAJOR CONTRIBUTIONS. IN THE FIRST, we advanced the state-of-the-art in building libraries of polymorphic structures for generic programming in COQ.

Previous models used *dependent records* to structure programming and emulate the notion of telescope — the clear type-theoretic incarnation of mathematical structures. They composed them to form hierarchies of structures, just as mathematical structures themselves are naturally organized along hierarchies: groups are defined starting from monoids, rings from groups, etc ... Their organization paradigm depended on the various products inherent in the calculus of COQ: they used parametric arguments (Pebble-style records) or dependent arguments (telescopic records). Those products facilitated the proliferation of copies of arguments within a term, with little-to-no sharing.

We proposed a new paradigm, Packed Classes, which also structures hierarchies using those products, but packed within inductive types. This let us use the fact that COQ maintains an environment of inductives when analyzing terms and tries to deal with them by reference as much as possible. We measured this sharing made COQ behave as if it was manipulating terms much smaller than in the other paradigms, and demonstrated we lost none of the expressivity. This achievement was prompted by a contribution to a large formal development making extensive use of the generic programming facilities of COQ, notably `Canonical Structures`, a flavor of type classes. It was, then, implemented within that library and represented a major step forward in the usability of this generic programming construct, after the porting of several tens of thousands of lines of code.

THE SECOND CONTRIBUTION OF THIS WORK is a method for imperatively triggering the instantiation mechanism of `Canonical Structures`, with custom equational constraints. It makes a synergical use of phantom types and notations. It solves a key problem of expressivity in COQ: Mathematicians, as well as COQ users, like to refer to mathematical objects just by mentioning the essential computational part of their definition. For example, when a mathematician speaks of the image of x by $f \circ g$, he means sometimes the composition of the *functions* f and g and, sometimes the composition of the *homomorphisms* f and g. The audience is supposed to deduce which from the context -- for example, there is a strong hint that we are talking about *homomorphisms* if $x = 1$. But composition $(\circ)$ is defined computationally even if f and g are not morphisms.

On the other hand, `Canonical Structure`s are a way to peg some data on Coq terms. Here, we would use them to peg some mathematical structure on mathematical objects : morphisms on functions. But the computational part of mathematical object must, for that, exist *independently* of the existence of the structure of its components. When it does not — as, for example, when talking about the domain of f — `Canonical Structure`s are of no help and the user has to provide the whole structure "by hand". This contribution provides the user with the ability to manipulate expressions such as the *domain* of a morphism without having to enter more than a reference to f, the underlying function — that is, the essential computational part of the equation. As an ancillary but valuable effect, we have obtained a way to test (in the software-engineering sense) the sometimes mercurial inferability of `Canonical Structure`s.

THE THIRD CONTRIBUTION consists in a generic treatment of a frequent kind of subgroup definition in mathematics. Recognizing that those definitions, when represented as terms of type theory, yield polymorphic *subgroup-defining functions* , we give an abstraction for this notion and give them a unified treatment in the formalization. Our work shows that the *generic* approach in representing those varied mathematical definitions —-something which has been mostly forgotten in mathematics since the height of the Soviet era — provides a way to shorten menial proofs using compositionality. Moreover, we were able to render in formal terms the link between those subgroup-defining functions and some common group properties, represented as *classes of groups*. We provided a way to study the one using the other within our formal library. We finish with an exploration of the relationship between those polymorphic terms and the well-known properties of relational parametricity.

FUTURE WORK in generic programming for mathematical formalization in Coq would involve, to start with, the improvement of the prover itself. It will provide a unification of the two flavors of type classes (`Canonical Structure`s and `Class`es) and, hopefully, better language support for manipulating the higher order unification algorithm. In particular, we are envious of the "hints in unification" approach of Matita, but would also like to see the instance chains described in § 1.4.4 on p. 74 become native constructs of the language.

The size and consequent performance issues of our library make us wish for a more efficient back-end for dependently typed programming with records. We wish for — as an example — the special treatment of polymorphic of record projections recently found to yield a great performance increase in Agda.

Finally, another area in which further research needs to be done is in the development of a reflection method capturing the full strength of the relational parametricity of a given λ-calculus, within the language, scalable along the ongoing development of the theory of parametricity for various calculi — from system F to, hopefully, a calculus relatively close to that of Coq— that the literature will certainly continue to show.

Bibliography

M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '93*, number X, pages 157–170, New York, New York, USA, 1993. ACM Press. ISBN 0897915607. doi:10.1145/158511.158622.

M. Abbott. *Categories of containers*. Ph.d. thesis, University of Leicester, 2003. URL http://www.cs.le.ac.uk/people/ma139/docs/thesis.pdf.

M. Abbott, T. Altenkirch, and N. Ghani. Categories of Containers. In A. Gordon, editor, *Foundations of Software Science and Computation Structures*, volume 2620 of *Lecture Notes in Computer Science*, pages 23–38. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-00897-2. doi:10.1007/3-540-36576-1_2.

P. Aczel. Galois: a theory development project. 1993. URL http://www.cs.manchester.ac.uk/~petera/galois.ps.gz.

P. Aczel. Simple overloading for type theories. In *Types for proofs and programs*, 1994a. URL http://www.cs.man.ac.uk/~petera/overloading-for-type-theories-1994.pdf.

P. Aczel. A notion of class for theory development in algebra in a predicative type theory. In *Talk presented at the workshop on Types for Proofs and Programs, Båstad, Sweden*, 1994b. URL http://www.cs.man.ac.uk/~petera/classes-for-theory-development.pdf.

P. Aczel and G. Barthe. A notion of class for Type Theory. 1993. URL http://www.cs.man.ac.uk/~petera/classes-for-type-theory-1993.pdf.

T. Altenkirch. *Constructions, Inductive Types and Strong Normalization*. Ph.d. thesis, University of Edimburgh, Nov. 1993. URL http://www.lfcs.inf.ed.ac.uk/reports/93/ECS-LFCS-93-279/.

S. A. Amitsur. A General Theory of Radicals. I. Radicals in Complete Lattices. *American Journal of Mathematics*, 74(4):774, Oct. 1952. ISSN 00029327. doi:10.2307/2372225.

S. A. Amitsur. A General Theory of Radicals. III. Applications. *American Journal of Mathematics*, 76(1):126, Jan. 1954a. ISSN 00029327. doi:10.2307/2372404.

S. A. Amitsur. A General Theory of Radicals. II. Radicals in Rings and Bicategories. *American Journal of Mathematics*, 76(1):100, Jan. 1954b. ISSN 00029327. doi:10.2307/2372403.

K. Appel and W. Haken. The Solution of the Four-Color-Map Problem. *Scientific American*, 237(4):108–121, Oct. 1977. ISSN 0036-8733. doi:10.1038/scientificamerican1077-108.

R. D. Arthan. Some Mathematical Case Studies in ProofPower-HOL. Technical Report January, 2006a. URL http://www.lemma-one.com/papers/51.pdf.

R. D. Arthan. Mathematical Case Studies: — Some Group Theory. Technical Report July, 2006b. URL http://www.lemma-one.com/ProofPower/examples/wrk068.pdf.

A. Asperti, W. Ricciotti, C. Sacerdoti Coen, and E. Tassi. Hints in Unification. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 84–98. Springer Berlin / Heidelberg, 2009. doi:10.1007/978-3-642-03359-9_8.

D. Aspinall and M. Hofmann. Dependent Types. In B. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 2, pages 45–86. MIT press, 2005. ISBN 0262162288. URL http://homepages.inf.ed.ac.uk/da/attapl/.

M. H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998. ISBN 0-201-30956-4.

B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 3–15, New York, New York, USA, 2008. ACM New York, NY, USA. ISBN 9781595936899. doi:10.1145/1328438.1328443.

R. Baer. Group Theoretical Properties and Functions. *Colloquium Mathematicum*, 14:285–328, 1966. URL http://journals.impan.gov.pl/cgi-bin/shvold?cm14.

A. Bailey. *The machine-checked literate formalisation of algebra in type theory*. Ph.d. thesis, University of Manchester, 1998a. URL http://anthonybailey.net/thesis/.

A. Bailey. Coercion synthesis in computer implementations of type-theoretic frameworks. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs*, volume 1512 of *Lecture Notes in Computer Science*, pages 9–27. Springer Berlin / Heidelberg, 1998b. doi:10.1007/BFb0097784.

H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science*, volume 2, chapter 2, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992. ISBN 0198537611.

B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Denis Diderot (Paris VII), 1999. URL http://www.lix.polytechnique.fr/~barras/publi/these_barras.ps.gz.

B. Barras and B. Grégoire. On the Role of Type Decorations in the Calculus of Inductive Constructions. In L. Ong, editor, *Computer Science Logic*, volume 3634 of *Lecture Notes in Computer Science*, pages 151–166. Springer Berlin / Heidelberg, 2005. doi:10.1007/11538363_12.

G. Barthe. Implicit coercions in type systems. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin / Heidelberg, 1996. doi:10.1007/3-540-61780-9_58.

G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. *Journal of Functional Programming*, 13(02):261–293, Mar. 2003. ISSN 0956-7968. doi:10.1017/S0956796802004501.

H. Bender and G. Glauberman. *Local Analysis for The Odd Order Theorem (London Mathematical Society Lecture Note Series, No 188)*. Cambridge University Press, 1995. ISBN 0521457165. doi:10.2277/0521457165.

J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming - ICFP '10*, number Section 4, page 345, New York, New York, USA, 2010. ACM Press. ISBN 9781605587943. doi:10.1145/1863543.1863592.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions*. Springer-Verlag, 2004. ISBN 3540208542.

Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. Canonical Big Operators. In O. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 86–101. Springer Berlin / Heidelberg, 2008. doi:10.1007/978-3-540-71067-7_11.

G. Betarte and A. Tasistro. Extension of Martin-Löf's type theory with record types and subtyping. In G. Sambin and J. M. Smith, editors, *Twenty-Five Years of Constructive Type Theory, Proceedings of a Congress held in Venice, October 1995*, chapter 2, pages 21–39. Oxford University Press, 1998. ISBN 0198501277.

F. Biancuzzi and S. Warden. *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. O'Reilly Media, Inc., Beijing, 2009. ISBN 0596515170, 9780596515171.

L. Birkedal and R. Mø gelberg. On the definition of parametricity. Technical report, IT University of Copenhagen, Copenhagen, 2004. URL https://itu.dk/en/Forskning/Technical-Reports/2004/On-the-Definition-of-Parametricity.

E. Bishop. *Foundations of constructive analysis*. McGraw-Hill Book Co., New York, 1967.

N. Bourbaki. *Théorie des ensembles*. Springer, 2006. ISBN 3540340343.

A. Bove, A. Krauss, and M. Sozeau. Partiality and Recursion in Interactive Theorem Provers - An Overview. *Under consideration for publication in Math. Struct. in Comp. Science*, 2011.

E. Brady, C. McBride, and J. McKinna. Inductive Families Need Not Store Their Indices. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-22164-7. doi:10.1007/978-3-540-24849-1_8.

R. M. Burstall. Programming with modules as typed functional programming. In Shin Sedai Konpyūta Gijutsu Kaihatsu Kikō, editor, *Fifth generation computer systems 1984*, pages 103–112. OHMSHA Ltd. Tokyo and North-Holland, Tokyo, Japan, 1984. ISBN 0444876731.

P. Callaghan. Coherence Checking of Coercions in Plastic. In *APPSEM Workshop on Subtyping & Dependent Types in Programming*, pages 1–20, Ponte de Lima, Portugal, 2000. URL http://www-sop.inria.fr/oasis/DTP00/Proceedings/callaghan.ps.

P. Callaghan and Z. Luo. Implementation Techniques for Inductive Types in Plastic. In T. Coquand, P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs*, volume 1956 of *Lecture Notes in Computer Science*, pages 245–262. Springer Berlin / Heidelberg, 2000. doi:10.1007/3-540-44557-9_6.

V. Capretta. A polymorphic representation of induction-recursion. Technical report, ICIS, Radboud University Nijmegen, 2004. URL http://www.cs.ru.nl/~venanzio/publications/induction_recursion.pdf.

M. M. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. *ACM SIGPLAN Notices*, 40 (1):1–13, Jan. 2005. ISSN 03621340. doi:10.1145/1047659.1040306.

G. Chen. Coercive subtyping for the calculus of constructions. *ACM SIGPLAN Notices*, 38(1):150–159, Jan. 2003. ISSN 03621340. doi:10.1145/640128.604145.

J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, Ithaca, NY, 2003. URL http://hdl.handle.net/1813/5614.

A. Chlipala. Certified Programming with Dependent Types. 2009. URL http://adam.chlipala.net/cpdt/.

C. Chong, Y. Leong, and J. Serre. An interview with Jean-Pierre Serre. *The Mathematical Intelligencer*, 8(4):8–13, 1986. ISSN 0343-6993. doi:10.1007/BF03026112.

C. Cohen. Les types quotient en Coq. Master's thesis, Master Parisien de Recherche en Informatique, Université Denis Diderot (Paris VII), August 2010. URL http://perso.crans.org/~cohen/stageM2/.

W. R. Cook. *On understanding data abstraction, revisited*, volume 44. ACM Press, New York, New York, USA, 2009. ISBN 9781605587660. doi:10.1145/1640089.1640133.

The Coq Proof Assistant Reference Manual, version 8.3. Coq development team, 2010. URL http://coq.inria.fr/distrib/V8.3/refman/.

T. Coquand. An Analysis of Girard's Paradox. In *LICS*, pages 227–236. IEEE Computer Society, 1986. URL http://www.cse.chalmers.se/~coquand/paradox.ps.

T. Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1-3):167–177, May 1996. ISSN 01676423. doi:10.1016/0167-6423(95)00021-6.

C. Cornes. *Conception d'un langage de haut niveau de représentation de preuves : Récurrence par filtrage de motifs Unification en présence de types inductifs primitifs Synthèse de lemmes d'inversion*. Thèse de doctorat, Université Denis Diderot (Paris VII), 1997. URL http://www.fing.edu.uy/~cornes/Papers/These.ps.

J. Courant. Explicit Universes for the Calculus of Constructions. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 2410 of *Lecture Notes in Computer Science*, pages 95–145. Springer Berlin / Heidelberg, 2002. doi:10.1007/3-540-45685-6_9.

L. Cruz-Filipe, H. Geuvers, and F. Wiedijk. C-CoRN, the Constructive Coq Repository at Nijmegen. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Mathematical Knowledge Management*, volume 3119 of *Lecture Notes in Computer Science*, pages 88–103. Springer Berlin / Heidelberg, 2004. doi:10.1007/978-3-540-27818-4_7.

N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91(2):189–204, Apr. 1991. ISSN 08905401. doi:10.1016/0890-5401(91)90066-B.

S. E. Dickson. A torsion theory for Abelian categories. *Transactions of the American Mathematical Society*, 121(1): 223–223, Jan. 1966. ISSN 0002-9947. doi:10.1090/S0002-9947-1966-0191935-0.

J. D. Dixon. *Problems in group theory*. Courier Dover Publications, 1973. ISBN 048661574X.

G. Dowek. A Complete Proof Synthesis Method for the Cube of Type Systems. *Journal of Logic and Computation*, 3(3):287–315, 1993. ISSN 0955-792X. doi:10.1093/logcom/3.3.287.

G. Dowek. Higher-Order Unification and Matching. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1009–1062. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9, 0-262-18223-8.

D. Dreyer, R. Harper, M. M. T. Chakravarty, and G. Keller. Modular type classes. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '07*, page 63, New York, New York, USA, 2007. ACM Press. ISBN 1595935754. doi:10.1145/1190216.1190229.

M. du Sautoy. Thompson and Tits win the Abel Prize 2008. Technical report, Norwegian Academy of Science and Letters, 2008. URL http://www.abelprisen.no/en/prisvinnere/2008/marcus/index.html.

D. S. Dummit and R. M. Foote. *Abstract algebra*. Wiley, 2004. ISBN 9780471452348.

P. Dybjer. *Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics*, pages 280–306. Cambridge University Press, New York, NY, USA, 1991. ISBN 0521413001.

S. Eilenberg and S. MacLane. Natural Isomorphisms in Group Theory. *Proceedings of the National Academy of Sciences*, 28(12):537–543, 1942. URL https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1078535/.

S. Eilenberg and S. MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58(2):231–231, 1945. ISSN 0002-9947. doi:10.1090/S0002-9947-1945-0013131-6.

C. M. Elliott. *Extensions and Applications of Higher order Unification*. Ph.d. thesis, Carnegie Mellon University, 1990. URL http://conal.net/papers/elliott90.pdf.

W. Feit and J. G. Thompson. Solvability of groups of odd order. *Pacific Journal of Mathematics*, 13(3), 1963. URL http://projecteuclid.org/euclid.pjm/1103053941.

R. B. Findler and M. Flatt. *Modular object-oriented programming with units and mixins*. ACM Press, New York, New York, USA, 1998. ISBN 1581130244. doi:10.1145/289423.289432.

M. Fluet and R. Pucella. Practical Datatype Specializations with Phantom Types and Recursion Schemes. *Electronic Notes in Theoretical Computer Science*, 148(2):25, Oct. 2005. ISSN 15710661. doi:10.1016/j.entcs.2005.11.046.

M. Fluet and R. Pucella. Phantom types and subtyping. *Journal of Functional Programming*, 16(06):751, June 2006. ISSN 0956-7968. doi:10.1017/S0956796806006046.

A. A. Fraenkel, Y. Bar-Hillel, and A. Lévy. *Foundations of set theory*. Studies in logic and the foundations of mathematics. Noord-Hollandsche U.M., 1973. ISBN 9780720422702.

Y. Fujisawa, Y. Fuwa, and H. Shimizu. Euler's Theorem and Small Fermat's Theorem. *Journal of Formalized Mathematics*, 10(20), 1998. URL http://www.mizar.org/JFM/Vol10/euler_2.html.

R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17(02):145, Dec. 2006. ISSN 0956-7968. doi:10.1017/S0956796806006198.

B. J. Gardner. *Radical Theory*. Longman Scientific & Technical, Harlow, Essex, England and New York, 1989. ISBN 0470212713.

B. J. Gardner. Kurosh-Amitsur Radical Theory For Groups. *Analele Stiintifice ale Universitatii Ovidius Constanta, Seria Matematica*, 18(2):73–90, 2010. ISSN 0092-7872. URL http://www.emis.ams.org/journals/ASUO/volume-xviii-2010-fascicola-2.html.

F. Garillot. A small reflection on group automorphisms. Talk given at the TYPES 2008 conference, march 2008, Torino, Italy, 2008. URL http://www.garillot.net/types-slides.pdf.

F. Garillot and B. Werner. Simple Types in Type Theory: Deep and Shallow Encodings. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 368–382. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-74590-7. doi:10.1007/978-3-540-74591-4_27.

F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging Mathematical Structures. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer Berlin / Heidelberg, 2009. doi:10.1007/978-3-642-03359-9_23.

H. Geuvers. Induction Is Not Derivable in Second Order Dependent Type Theory. In S. Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 166–181. Springer Berlin / Heidelberg, 2001. ISBN 978-3-540-41960-0. doi:10.1007/3-540-45413-6_16.

H. Geuvers. A Constructive Algebraic Hierarchy in Coq. *Journal of Symbolic Computation*, 34(4):271–286, Oct. 2002. ISSN 07477171. doi:10.1006/jsco.2002.0552.

E. Giménez. Codifying guarded definitions with recursive schemes. In P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer Berlin / Heidelberg, 1995. doi:10.1007/3-540-60579-7_3.

G. Gonthier. A computer-checked proof of the four-colour theorem. 2005. URL http://research.microsoft.com/en-us/people/gonthier/4colproof.pdf.

G. Gonthier. Formal Proof – The Four Color Theorem. *Notices of the American Mathematical Society*, 55(11): 1382–1393, 2008. URL http://www.ams.org/notices/200811/tx081101382p.pdf.

G. Gonthier. Formalizing the structure of extremal p-groups. Talk given at the MAP 2010 workshop, november 2010, Logroño, Spain, 2010. URL http://wiki.portal.chalmers.se/cse/uploads/ForMathInternalPages/Georges-Nov2010.pdf.

G. Gonthier. Point-Free, Set-Free Concrete Linear Algebra. In M. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *Interactive Theorem Proving*, volume 6898 of *Lecture Notes in Computer Science*, pages 103–118. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-22862-9. doi:10.1007/978-3-642-22863-6_10.

G. Gonthier and S. Le Roux. An Ssreflect Tutorial. Rapport Technique RT-0367, INRIA, 2009. URL http://hal.inria.fr/inria-00407778/en/.

G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010. URL http://jfr.cib.unibo.it/article/view/1979.

G. Gonthier, A. Mahboubi, L. Rideau, E. Tassi, and L. Théry. A Modular Formalisation of Finite Group Theory. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 86–101. Springer Berlin / Heidelberg, 2007. doi:10.1007/978-3-540-74591-4_8.

G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008. URL http://hal.inria.fr/inria-00258384/en/.

G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional programming - ICFP '11*, page 163, New York, New York, USA, 2011. ACM Press. ISBN 9781450308656. doi:10.1145/2034773.2034798.

D. Gorenstein. *Finite groups*. American Mathematical Society, second edition, 2007. ISBN 0821843427.

B. Grégoire and A. Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-28372-0. doi:10.1007/11541868_7.

D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. *Concepts: linguistic support for generic programming in C++*. ACM Press, New York, New York, USA, 2006. ISBN 1595933484. doi:10.1145/1167473.1167499.

E. L. Gunter. Doing algebra in simple type theory. Technical report, University of Pennsylvania, Philadelphia, 1989. URL http://repository.upenn.edu/cis_reports/789.

F. Haftmann and M. Wenzel. Constructive Type Classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 160–174. Springer Berlin / Heidelberg, 2007. doi:10.1007/978-3-540-74464-1_11.

T. Hallgren. Fun with functional dependencies. In *Proc Joint CS/CE Winter Meeting, Chalmers Univerity, Varberg, Sweden*, 2001. URL http://www.cs.chalmers.se/Cs/wm-01/hallgren.ps.

MATITA User Manual. The HELM team, 2010. URL http://matita.cs.unibo.it/documentation.shtml.

H. Herbelin. Type inference with algebraic universes in the Calculus of Inductive Constructions. 2005. URL http://yquem.inria.fr/~herbelin/publis/univalgcci.pdf.

H. Herbelin. The rules of PCIC. Email on the coq-club mailing-list, November 2009. URL https://sympa-roc.inria.fr/wws/arc/coq-club. Msg-id: <20091102214459.A17165@pauillac.inria.fr>.

R. Hinze. Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The fun of programming*, chapter 12, pages 245–262. Palgrave McMillan, 2003. ISBN 0333992857.

C. A. R. Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2):75–83, Feb. 1981. ISSN 00010782. doi:10.1145/358549.358561.

J. G. Hook and D. J. Howe. Impredicative Strong Existential Equivalent to Type:Type. Technical report, Cornell University, Ithaca, NY, USA, 1986. URL http://www.ecommons.cornell.edu/handle/1813/6600.

G. Huet. A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, 1(1):27–57, June 1975. ISSN 03043975. doi:10.1016/0304-3975(75)90011-0.

G. Huet. The Constructive Engine. In R. Narasimhan, editor, *A perspective in Theoretical Computer Science. Commemorative Volume for Gift Siromoney*. World Scientific Publishing, 1989. ISBN 9971509253.

G. Huet. Higher Order Unification 30 Years Later. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 2410 of *Lecture Notes in Computer Science*, pages 241–258. Springer Berlin / Heidelberg, 2002. doi:10.1007/3-540-45685-6_2.

P. Jackson. Exploring abstract algebra in constructive type theory. *Automated Deduction—CADE-12*, (July 1994): 1–15, 1994. doi:10.1007/3-540-58156-1_43.

P. Jackson. *Enhancing the Nuprl proof-development system and applying it to computational abstract algebra*. Ph.d. thesis, Cornell University, 1995. URL https://ecommons.library.cornell.edu/handle/1813/7167.

P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '04*, pages 99–110, New York, New York, USA, 2004. ACM Press. ISBN 158113729X. doi:10.1145/964001.964010.

M. Jones. Type Classes with Functional Dependencies. In G. Smolka, editor, *Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer Berlin / Heidelberg, 2000. doi:10.1007/3-540-46425-5_15.

M. P. Jones. Computing with lattices: An application of type classes. *Journal of Functional Programming*, 2(04):475, Nov. 1992. ISSN 0956-7968. doi:10.1017/S0956796800000514.

M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(01):1, Nov. 1993. ISSN 0956-7968. doi:10.1017/S0956796800001210.

W. Kahl and J. Scheffczyk. Named instances for Haskell type classes. In *Haskell Workshop*, 2001. URL http://www.cas.mcmaster.ca/~kahl/Publications/Conf/Kahl-Scheffczyk-2001.html.

D. Kapur, D. R. Musser, and A. A. Stepanov. *Operators and algebraic structures*. ACM Press, New York, New York, USA, 1981. ISBN 0897910605. doi:10.1145/800223.806763.

O. Kiselyov. Operator Overloading. Email on the ocaml mailing-list, March 2007. URL http://okmij.org/ftp/ML/ML.html#typeclass.

O. Kiselyov and R. Lämmel. Haskell's overlooked object system. *Draft*, (September), 2005. URL http://homepages.cwi.nl/~ralf/OOHaskell/.

O. Kiselyov and C.-c. Shan. Functional pearl : Implicit configurations. In *Proceedings of the ACM SIGPLAN workshop on Haskell - Haskell '04*, page 33. ACM Press, New York, New York, USA, 2004. ISBN 1581138504. doi:10.1145/1017472.1017481.

G. Klein, M. Norrish, T. Sewell, H. Tuch, S. Winwood, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, and R. Kolanski. seL4: formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107, June 2010. ISSN 00010782. doi:10.1145/1743546.1743574.

K. Knight. Unification: a multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, Mar. 1989. ISSN 03600300. doi:10.1145/62029.62030.

N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. *Design patterns in separation logic*. ACM Press, New York, New York, USA, 2008. ISBN 9781605584201. doi:10.1145/1481861.1481874.

A. G. Kuroš. Radicals of rings and algebras. In *Colloq. Math. Soc. Janos Bolyai, Vol. 6*, pages 297–314, Amsterdam, 1973a. North-Holland. URL http://www.ams.org/mathscinet-getitem?mr=345998.

A. G. Kuroš. Radicals in the theory of groups. In *Colloq. Math. Soc. Janos Bolyai, Vol. 6*, pages 271–296, Amsterdam, 1973b. North-Holland. URL http://www.ams.org/mathscinet-getitem?mr=345997.

H. Kurzweil and B. Stellmacher. *The theory of finite groups: an introduction*. Springer, 2004. ISBN 0387405100.

R. Lämmel and K. Ostermann. Software extension and integration with type classes. *Proceedings of the 5th international conference on Generative programming and component engineering - GPCE '06*, page 161, 2006. doi:10.1145/1173706.1173732.

M. Lecat. *Erreurs des Mathématiciens des origines à nos jours*. Ancienne librairie Castaigne & librairie Ém. Desbarax, Bruxelles, Louvain, 1935.

G. Lee and B. Werner. A proof-irrelevant model of CIC with predicative induction and judgemental equality. *to appear in Logical Methods in Computer Science*, 2011. URL http://www.lmcs-online.org/.

D. Leijen and E. Meijer. *Domain specific embedded compilers*, volume 35. ACM Press, New York, New York, USA, 1999. ISBN 1581132557. doi:10.1145/331960.331977.

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107, July 2009. ISSN 00010782. doi:10.1145/1538788.1538814.

P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. Thèse de doctorat, Université Paris-Sud (Paris XI), July 2004. URL http://tel.archives-ouvertes.fr/tel-00150912/en/.

R. Lidl and G. Pilz. *Applied abstract algebra*. Springer, New York, 1998. ISBN 0387982906.

G. Lo Russo. An Interview with A. Stepanov, June 2000. URL http://www.stlport.org/resources/StepanovUSA.html. STLport, Edizioni Infomedia srl.

J. O. López Gerena. *Closure operators, torsion theories, and radicals in a non-abelian environment*. Master of science thesis, University of Puerto Rico-Mayaguez, 2009. URL http://gradworks.umi.com/14/68/1468669.html.

Z. Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, 1994. ISBN 0198538359.

Z. Luo. Coercive subtyping in type theory. In D. van Dalen and M. Bezem, editors, *Computer Science Logic*, volume 1258 of *Lecture Notes in Computer Science*, pages 275–296. Springer Berlin / Heidelberg, 1997. doi:10.1007/3-540-63172-0_45.

Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, Feb. 1999. ISSN 0955-792X. doi:10.1093/logcom/9.1.105.

S. Mac Lane and G. Birkhoff. *Algebra*. Chelsea Pub. Co., New York, 1988. ISBN 0828403309.

S. MacLane. Duality for groups. *Bulletin of the American Mathematical Society*, 56(6):485–517, Nov. 1950. ISSN 0002-9904. doi:10.1090/S0002-9904-1950-09427-0.

D. MacQueen. *Modules for standard ML*. ACM Press, New York, New York, USA, 1984. ISBN 0897911423. doi:10.1145/800055.802036.

L. Márki. The Categorical Approach to General Radical Theory - a survey. In E. Puczylowski, editor, *Talk presented at the workshop on Radicals of rings and related topics, Stefan Banach International Center, 2-8 August*, Warsaw, Poland, 2009. URL http://aragorn.pb.bialystok.pl/~piotrgr/BanachCenter/lectures/Marki.pdf.

J.-P. Marquis. What is category theory? In G. Sica, editor, *What is category theory ?*, chapter 9, pages 221–255. Polimetrica s.a.s., 2006. ISBN 8876990313.

P. Martin-Löf. *An intuitionistic theory of types*. Bibliopolis, 1984. ISBN 88-7088-105-9.

C. McBride. Faking it (Simulating dependent types in Haskell). *Journal of Functional Programming*, 12(4-5):375–392, July 2003. ISSN 0956-7968. doi:10.1017/S0956796802004355.

D. Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation*, 1(4):497–536, 1991. ISSN 0955-792X. doi:10.1093/logcom/1.4.497.

A. Miquel. *Le Calcul Des Constructions Implicite: Syntaxe Et Sémantique*. Thèse de doctorat, Université Denis Diderot (Paris VII), 2001. URL http://perso.ens-lyon.fr/alexandre.miquel/publis/these.pdf.

J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(3):470–502, 1988. doi:10.1145/44501.45065.

J. G. Morris and M. P. Jones. Instance chains. *ACM SIGPLAN Notices*, 45(9):375, Sept. 2010. ISSN 03621340. doi:10.1145/1932681.1863596.

S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of Programming Using Dependent Types. In P. Audebaud and C. Paulin-Mohring, editors, *Mathematics of Program Construction*, volume 5133 of *Lecture Notes in Computer Science*, pages 268–283. Springer Berlin / Heidelberg, 2008. doi:10.1007/978-3-540-70594-9_15.

O. Müller and K. Slind. Treating Partiality in a Logic of Total Functions. *The Computer Journal*, 40(10):640–651, Oct. 1997. ISSN 0010-4620. doi:10.1093/comjnl/40.10.640.

U. Norell. *Towards a practical programming language based on dependent type theory*. Ph.d. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, Sept. 2007. URL https://publications.lib.chalmers.se/cpl/record/index.xsql?pubid=46311.

U. Norell. Agda performance improvements. Email on the agda mailing-list, August 2011. URL https://lists.chalmers.se/pipermail/agda/2011/003266.html. Msg-id: <CAJjNqYGxDDNtnO8247u=GMx1e1pxKkVCYdtYb0XTK=MZUQn3yQ@mail.gmail.com>.

M. Odersky and A. Moors. Fighting Bit Rot with Types (Experience Report: Scala Collections). *FSTTCS 2009*, page 427, 2009. doi:10.4230/LIPIcs.FSTTCS.2009.2338.

M. Odersky and M. Zenger. Independently Extensible Solutions to the Expression Problem. In *Proc. FOOL 12*, Jan. 2005. URL https://infoscience.epfl.ch/record/64421.

K. Ohtake. A torsion theory for the category of finite groups. *Gumma University technical report*, 56:5–8, 2008. ISSN 0017-5668. URL http://hdl.handle.net/10087/3001.

B. Oliveira. Modular Visitor Components. In S. Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 269–293. Springer Berlin / Heidelberg, 2009. doi:10.1007/978-3-642-03013-0_13.

B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '10*, page 341, New York, New York, USA, 2010. ACM Press. ISBN 9781450302036. doi:10.1145/1869459.1869489.

S. Ould Biha. *Composants mathématiques pour la théorie des groupes*. Thèse de doctorat, Université de Nice - Sophia Antipolis, 2010. URL http://tel.archives-ouvertes.fr/tel-00493524/en.

C. Paulin-Mohring. Inductive definitions in the system Coq: rules and properties. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer Berlin / Heidelberg, 1993. doi:10.1007/BFb0037116.

C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard (Lyon I), Dec. 1996. URL http://www.lri.fr/~paulin/PUBLIS/habilitation.ps.gz.

A. J. Perlis. Special Feature: Epigrams on programming. *ACM SIGPLAN Notices*, 17(9):7–13, Sept. 1982. ISSN 03621340. doi:10.1145/947955.1083808.

T. Peterfalvi. *Character Theory for the Odd Order Theorem (London Mathematical Society Lecture Note Series, No 272)*. Cambridge University Press, 2000. ISBN 9780521646604. doi:10.2277/052164660X.

S. Peyton-Jones. The Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13 (01), Jan. 2003. ISSN 0956-7968. doi:10.1017/S0956796803000315.

S. Peyton-Jones, M. P. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, 1997. URL http://research.microsoft.com/en-us/um/people/simonpj/Papers/type-class-design-space/.

F. Pfenning. *Unification and anti-unification in the calculus of constructions*. IEEE Comput. Sco. Press, 1991. ISBN 0-8186-2230-X. doi:10.1109/LICS.1991.151632.

B. I. Plotkin. The functorials, radicals and coradicals in groups. *Ural. Gos. Univ. Mat. Zap.*, 7(3):150–182, 1969. URL http://www.ams.org/mathscinet-getitem?mr=285614.

B. I. Plotkin. Radicals in groups, operations on classes of groups, and radical classes. *American Mathematical Society Translations*, 119(2):205–244, 1983.

G. Plotkin and M. Abadi. A logic for parametric polymorphism. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer Berlin / Heidelberg, 1993. ISBN 978-3-540-56517-8. doi:10.1007/BFb0037118.

R. Pollack. Implicit syntax. In *Informal Proceedings of First Workshop on Logical Frameworks, Antibes*, pages 421–434, Sept. 1990. URL http://www.lfcs.inf.ed.ac.uk/research/types-bra/proc/proc90.ps.gz.

R. Pollack. Typechecking in pure type systems. In *Informal Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden*, pages 271–288, 1992. URL http://www.lfcs.inf.ed.ac.uk/research/types-bra/proc/proc92.ps.gz.

R. Pollack. Dependently Typed Records for Representing Mathematical Structure. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 462–479. Springer Berlin / Heidelberg, 2000. doi:10.1007/3-540-44659-1_29.

R. Pollack. Dependently Typed Records in Type Theory. *Formal Aspects of Computing*, 13(3):386–402, 2002. ISSN 0934-5043. doi:10.1007/s001650200018.

F. Pottier and D. Rémy. The Essence of ML Type Inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005. ISBN 0-262-16228-8. URL http://cristal.inria.fr/attapl/.

R. Prince, N. Ghani, and C. McBride. Proving Properties about Lists Using Containers. In J. Garrigue and M. Hermenegildo, editors, *Functional and Logic Programming*, volume 4989 of *Lecture Notes in Computer Science*, pages 97–112. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-78968-0. doi:10.1007/978-3-540-78969-7_9.

D. Pym. *Proofs, Search and Computation in General Logic*. Ph.d. thesis, University of Edimburgh, 1990. URL http://www.lfcs.inf.ed.ac.uk/reports/90/ECS-LFCS-90-125/.

J. C. Reynolds. Types, Abstraction and Parametric Polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, September 19-23, 1983*, volume 83, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb. 1978. ISSN 00010782. doi:10.1145/359340.359342.

J. Rotman. *An introduction to the theory of groups*. Springer, 1995. ISBN 0387942858.

P. Rudnicki. Commutative Algebra in the Mizar System. *Journal of Symbolic Computation*, 32(1-2):143–169, July 2001. ISSN 07477171. doi:10.1006/jsco.2001.0456.

C. Sacerdoti Coen. A Semi-reflexive Tactic for (Sub-)Equational Reasoning. In J.-C. Filliâtre, C. Paulin-Mohring, and B. Werner, editors, *Types for Proofs and Programs*, volume 3839 of *Lecture Notes in Computer Science*, pages 98–114. Springer Berlin / Heidelberg, 2006. doi:10.1007/11617990_7.

C. Sacerdoti Coen and E. Tassi. Working with Mathematical Structures in Type Theory. In M. Miculan, I. Scagnetto, and F. Honsell, editors, *Types for Proofs and Programs*, volume 4941 of *Lecture Notes in Computer Science*, pages 157–172. Springer Berlin / Heidelberg, 2008. doi:10.1007/978-3-540-68103-8_11.

A. Saibi. Typing algorithm in type theory with inheritance. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 292–301, New York, New York, USA, 1997. ACM New York, NY, USA. ISBN 0897918533. doi:10.1145/263699.263742.

A. Saibi. *Outils Génériques de modélisation et de démonstration pour la Formalisation des Mathématiques en théorie des Types, Application à la théorie des catégories*. Thèse de doctorat, Université Pierre et Marie Curie (Paris VI), 1999. URL http://tel.archives-ouvertes.fr/tel-00523810.

A. Saibi and G. Huet. Constructive category theory. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, language, and interaction*, volume 20, pages 239—-275. M.I.T. Press, Cambridge, MA, USA, Jan. 2000. ISBN 0-262-16188-5.

H. Saidi. Résolution d'équations dans le système T de Gödel. Master's thesis, DEA d'Informatique Fondamentale, Université Denis Diderot (Paris VII), Sept. 1994. URL http://www.csl.sri.com/users/saidi/PAPERS/dea94.html.

T. Santen. Isomorphisms — A Link Between the Shallow and the Deep. In Y. Bertot, G. Dowek, L. Théry, A. Hirschowitz, and C. Paulin, editors, *Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, page 840. Springer Berlin / Heidelberg, 1999. doi:10.1007/3-540-48256-3_4.

L. Scott, R. Solomon, J. Thompson, J. Walter, and E. Zelmanov. Walter Feit (1930–2004). *Notices of the American Mathematical Society*, 52(7):728–735, 2005. URL http://www.ams.org/notices/200507/fea-feit.pdf.

D. Seidel and J. Voigtländer. Automatically Generating Counterexamples to Naive Free Theorems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 175–190. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-12250-7. doi:10.1007/978-3-642-12251-4_14.

J. Shapiro, S. Sridhar, and S. Doerrie. BitC language specification. Technical report, The EROS Group, LLC, 2008. URL http://www.bitc-lang.org/docs/bitc/.

V. Siles. *Investigation on the typing of equality in type systems*. Thèse de doctorat, École Polytechnique, 2010. URL http://pastel.archives-ouvertes.fr/pastel-00556578/en/.

W. Snyder and J. Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(1-2):101–140, July 1989. ISSN 07477171. doi:10.1016/S0747-7171(89)80023-9.

E. Soubiran. *Modular development of theories and name-space management for the Coq proof assistant*. Thèse de doctorat, École Polytechnique, 2010.

M. Sozeau. *Un environnement pour la programmation avec types dépendants*. Thèse de doctorat, Université Paris-Sud (Paris XI), 2008. URL http://mattam.org/research/PhD.en.html.

M. Sozeau. A New Look At Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, 2009. URL http://jfr.cib.unibo.it/article/viewFile/1574/1077.

M. Sozeau and N. Oury. First-Class Type Classes. In O. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer Berlin / Heidelberg, 2008. doi:10.1007/978-3-540-71067-7_23.

B. Spitters and E. van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(04):795–825, July 2011. ISSN 0960-1295. doi:10.1017/S0960129511000119.

A. Spiwack. Followup (GT 3d of July) on reification. Email on the coqdev mailing-list, July 2011. URL https://sympa-roc.inria.fr/wws/arc/coqdev. Msg-id: <b3314ad40907210522o794f20d7vb2b35698b9b080d2@mail.gmail.com>.

A. Stepanov and M. Lee. The Standard Template Library. Technical report, Hewlett-Packard, Palo Alto, CA, 1995. URL http://www.hpl.hp.com/techreports/95/HPL-95-11.html.

J. Stillwell. *Elements of Algebra*. Springer-Verlag, New York Berlin Heidelberg, 1994. ISBN 0387942904.

C. Strachey. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, 13(1): 11–49, 2000. ISSN 1388-3690. doi:10.1023/A:1010000313106.

T. Streicher. *Semantical Investigations into Intensional Type Theory*. Habilitation thesis, Ludwig-Maximilians-Universität München, 1993. URL http://www.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf.

W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(04):423–436, 2008. ISSN 0956-7968. doi:10.1017/S0956796808006758.

I. Takeuti. The theory of parametricity in lambda cube. Technical report, Kyoto University Graduate School of Informatics, 2001. URL http://www.kurims.kyoto-u.ac.jp/~kyodo/kokyuroku/contents/pdf/1217-10.pdf.

E. Tassi. *Interactive Theorem Provers: issues faced as a user and tackled as a developer*. Dottorato di ricerca, Università di Bologna e Padova, 2008. URL http://www.cs.unibo.it/pub/TR/UBLCS/ABSTRACTS/2008.bib?ncstrl.cabernet//BOLOGNA-UBLCS-2008-03.

W. Verbruggen, E. de Vries, and A. Hughes. Polytypic programming in COQ. *Proceedings of the ACM SIGPLAN workshop on Generic programming - WGP '08*, page 49, 2008. doi:10.1145/1411318.1411326.

J. Voigtländer. Free theorems involving type constructor classes. *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming - ICFP '09*, page 173, 2009. doi:10.1145/1596550.1596577.

D. Vytiniotis and S. Weirich. Parametricity, type equality, and higher-order polymorphism. *Journal of Functional Programming*, 20(02):175, Apr. 2010. ISSN 0956-7968. doi:10.1017/S0956796810000079.

P. Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89*, number June, pages 347–359, New York, USA, 1989. ACM Press. ISBN 0897913280. doi:10.1145/99370.99404.

P. Wadler. The Expression Problem. Email on java-genericity mailing list, November 1998. URL http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.

P. Wadler. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1-3):201–226, May 2007. ISSN 03043975. doi:10.1016/j.tcs.2006.12.042.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89*, number October, pages 60–76, New York, USA, 1989. ACM Press. ISBN 0897912942. doi:10.1145/75277.75283.

S. Wehr and M. Chakravarty. ML Modules and Haskell Type Classes: A Constructive Comparison. In G. Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 188–204. Springer Berlin / Heidelberg, 2008. doi:10.1007/978-3-540-89330-1_14.

S. Wehr, R. Lämmel, and P. Thiemann. JavaGI : Generalized Interfaces for Java. In E. Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 347–372. Springer Berlin / Heidelberg, 2007. doi:10.1007/978-3-540-73589-2_17.

J. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1-3):111–156, June 1999. ISSN 01680072. doi:10.1016/S0168-0072(98)00047-5.

B. Werner. *Une théorie des constructions inductives*. Thèse de doctorat, Université Denis Diderot (Paris VII), 1994. URL http://tel.archives-ouvertes.fr/tel-00196524/.

J. Yallop. Practical generic programming in OCaml. In *Proceedings of the 2007 workshop on Workshop on ML - ML '07*, page 83, New York, New York, USA, 2007. ACM Press. ISBN 9781595936769. doi:10.1145/1292535.1292548.

X. Yu, A. Nogin, A. Kopylov, and J. Hickey. Formalizing abstract algebra in type theory with dependent records. In D. Basin and B. Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003) , Emerging Trends Proceedings*, pages 13–27. Universität Freiburg, 2003. URL http://files.metaprl.org/papers/formalaa.pdf.

OUTILS GÉNÉRIQUES DE PREUVE ET THËORIE DES GROUPES FINIS

Cette thèse présente des avancées dans l'utilisation des Structures Canoniques, un mécanisme du langage de programmation de l'assistant de preuve Coq, équivalent à la notion de classes de types. Elle fournit un nouveau modèle pour le développement de hiérarchies mathématiques à l'aide d'enregistrements dépendants, et, en guise d'illustration, fournit une reformulation de la preuve formelle de correction du cryptosystème RSA, offrant des méthodes de raisonnement algébrique ainsi que la représentation en théorie des types des notions mathématiques nécessaires (incluant les groupes cycliques, les groupes d'automorphisme, les isomorphismes de groupe). Nous produisons une extension du mécanisme d'inférence de Structures Canoniques à l'aide de types fantômes, et l'appliquons au traitement de fonctions partielles. Ensuite, nous considérons un traitement générique de plusieurs formes de définitions de sous-groupes rencontrées au long de la preuve du théorème de Feit-Thomspon, une large librairie d'algèbre formelle développée au sein de l'équipe Mathematical Components au laboratoire commun MSR-INRIA. Nous montrons qu'un traitement unifié de ces 16 sous-groupes nous permet de raccourcir la preuve de leur propriétés élémentaires, et d'obtenir des définitions offrant une meilleure compositionnalité. Nous formalisons une correspondance entre l'étude de ces fonctorielles, et des propriété de théorie des groupes usuelles, telles que représentées par la classe des groupes qui les vérifie. Nous concluons en explorant les possibilités d'analyse de la fonctorialité de ces définitions par l'inspection de leur type, et suggérons une voie d'approche vers l'obtention d'instances d'un résultat de paramétricité en Coq.

GENERIC PROOF TOOLS AND FINITE GROUP THEORY

This thesis presents advances in the use of Canonical Structures, a programming language construct of the Coq proof assistant equivalent to the notion of type classes. It provides a new model for developping hierarchies of mathematical structures using dependent records, and, as an illustration, reformulates the common formal proof of the correctness of the RSA cryptosystem, providing facilities for algebraic reasoning along with a formalization in type theory of the necessary mathematical notions (including cyclic groups, automorphism groups, group isomorphisms). We provide an extension of the Canonical Structure inference mechanism using phantom types, and apply it to treating the notion of partial functions. Next, we consider a generic treatment of several forms of subgroup definitions occurring in the formalization of the Feit-Thompson theorem, a large library of fomalized algebra developed in the Mathematical Components team at the MSR-INRIA joint laboratory. We show that a unified treatment of those 16 subgroups allows us to shorten menial proofs and obtain more composable definitions. We formalize a correspondence between the study of those group functorials, and some common and useful group-theoretic properties represented as the class of groups verifying them. We conclude in exploring the possibilities for analyzing the functoriality of those definitions by inspecting their type, and suggest a path towards obtaining instances of a parametricity result in Coq.