THE MACHINE-CHECKED LITERATE FORMALISATION OF ALGEBRA IN TYPE THEORY

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

January 1998

By

Anthony Bailey

Department of Computer Science

Contents

List of Figures

Abstract

I present a large-scale formalisation within a type theory of a proof of a result from abstract algebra. The formalisation body consists of files that are machine-checked to ensure their correctness, and also processed to produce a report on the proof that is human-readable. The resulting presentation is intended to approach being a standard informal account of some mathematics.

In addition to presenting this proof, the thesis also identifies and examines problems in reconciling the formal nature of the development with the wish for it to be easy to read. It presents some tools and methodologies for solving these problems, and discusses the advantages and disadvantages of these solutions. In particular, it addresses the implementation and use of implicit coercions within the type theory, the styles of proof that can be used, and the borrowing of concepts from the literate programming paradigm.

To be more specific, the algebra in question is a constructive version of the fundamental theorem of Galois Theory. The formalisation is developed within a variant of the Unified Theory of Types that is implemented by a modified version of the LEGO proof-checker.

Acknowledgements

First and foremost my thanks must go to Peter Aczel, who gave me far more of his time and energies than a research student could reasonably expect to get from a supervisor, and whose comments were always considered and insightful.

For the most part I did not have particularly close contact with researchers other than Peter, since we are the only two serious LEGO users here at Manchester. However my occasional forays outside to other establishments were very useful, and I always returned to my work with new enthusiasm after discussing it with other people. I'd therefore like to thank the types community in general, and to single out for special thanks Amokrane Saïbi and Zhaohui Luo. Amokrane persuaded me that it would not be as difficult as I feared for us to plunge into LEGO's source code and make the changes necessary to prototype our first ideas for classes and coercions, whilst Zhaohui provided a fresh and revealing perspective of coercions at the level of a logical framework, and helped form the idea of using coercions from unit types to implement overloading.

I must also thank the support staff here at Manchester for the use of their compute resources. LEGO likes to eat memory and processor time, and they were generous with both. My sincere thanks must also go to the staff of the department Computer Science building for putting up with me spending rather more of my time than I should there, due to my using it as a base for both my academic work and a rather active electronic social life.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Chapter 1

Introduction

This chapter provides an introduction to the areas explored by the thesis and the objectives I aimed to meet. It concludes with the traditional thesis outline, summarising the material that each following chapter will cover.

1.1 The main concepts

The title of this thesis is "The Machine-Checked Literate Formalisation of Algebra in Type Theory." I introduce the areas with which the thesis is concerned by talking about each of the terms used in the title in turn. The thesis will be based around the development of a particular piece of mathematics that is used to motivate investigation of the areas of interest and to act as a case-study for the methodologies and techniques under investigation.

1.1.1 Formalisation

Most mathematics is carried out in what in this thesis I shall refer to as an "informal" style. Informal mathematical arguments are often careful, and are intended to be non-ambiguous and fully consistent, because mathematics relies on exact communication. However informal mathematics is also commonly presented

in natural language, making much use of idioms and convention. Definitions of new objects may be introduced partially through analogy with existing constructs, or through an appeal to the intended semantics of the new object, and often proofs are merely sketched and details are omitted.

Formal mathematics is carried out within a framework of formal rules. Although the objects considered are designed to model objects and logical arguments with their own independent semantics, at base all construction and reasoning within the framework proceeds following simple syntactic rules.

There are several reasons for wanting to formalise a piece of mathematics. One is that, as long as one understands the meaning of a statement and trusts the formal system that it has been proven in, one is happier to believe that the statement has been proved. The precise statement of theorems and the application of named rules in the reasoning that proves them helps to avoid ambiguities in meaning and mismatches in argument that can lead to mistakes in informal mathematics. The extra work involved in making a particular formalisation may be of interest for its own sake, and the formalisation process in general is interesting from a philosophical standpoint.

However, working within a barebones formal system is very restrictive. Many of the advantages of the informal mathematical style are lost, such as natural language explanation, higher-level analogy, performing multiple logical steps at a time, pleasant notation with overloading, appeals to other works, and so on. Without the flexibility of the informal style, formalisations can become tediously long and painful to read. This flexibility is a double-edged sword; although dangerous in its ambiguity and potential for misunderstanding when abused, without it a formalisation can drown in all of the explicit detail that is introduced to prevent such ambiguity.

One of the beliefs underlying this thesis is that since the reasoning used in most

informal mathematics is consistent, it can be modelled within a formal framework; and that the mechanisms which make informal reasoning easier for human readers to follow are at some level unambiguously defined, and so can in principle be implemented through the use of a complicated enough combination of formal rules. The theme to my work in this thesis is to look at some ways of beginning to realise this principle. It attempts to narrow what I shall term *the formalisation gap* between a particular formal framework and informal mathematical practice.

1.1.2 Algebra

The algebra I formalised in the case-study is a constructive version of the fundamental theorem of Galois theory. The result itself forms a significant part of the material needed for the ongoing larger GALOIS project. An informal account of the relevant material is given in chapter 2.

GALOIS was started in 1992 at Manchester University by Peter Aczel as part of the "Types for Programs and Proofs" Esprit BRA project. Galois theory was chosen since it forms a large body of abstract algebra that makes use of results from many areas of mathematics. As such it is expected to provide experience of the many problems that must be solved to formalise "real" mathematics. One might expect to avoid in a smaller, more specialised "toy" development.

1.1.3 Type theory

I work within a variant of the Unified Theory of Types, or UTT, an extension of existing type theories by Zhaohui Luo. A newcomer to UTT may consider it to be a higher-order λ-calculus extended with some notions of definition and abbreviation, including inductive definition and rewriting, and with a hierarchy of type universes. UTT itself is impredicative, but I only make use of its predicative components. The resulting system has a strength similar to Martin-Löf type

theory with universes and inductive types. The environment is described in detail in chapter 3, and some further extensions to the expressiveness of the type theory through coercions are made in chapter 4.

1.1.4 Machine-checking

Since working within a formal system requires many careful and tedious manipulations of symbols under the application of various rules, using a computer to check that rules are correctly applied is an obvious idea.

Such *machine-checking* of proofs, where the computer verifies that a human has applied rules correctly, is distinct from the *automatic derivation* of proofs in a formal system, where the computer would itself look for a proof using a combination of heuristics and brute-force search.

In practice, most computer proof-assistants go some way beyond pure machine-checking. Since working in a formal system involves some repetitive sequences of rule applications, it is useful to automate some of these sequences. Information that is required for rule applications but that can be derived from the immediate context may also be checked by the machine without the need for human direction. This sort of computer assistance is necessary in order to make any large development of mathematics in a formal system a practical option.

The leaving of lower-level tasks to the machine may be seen as using automatic derivation on a very small scale, with the user now providing the direction for the formal proof at a slightly higher level. This is a small step in narrowing the formalisation gap between the original trusted formal system and the flexible informal style by using a machine as a translator. Much of this thesis involves further applications of this idea.

The UTT, the formal system in which I will work, is modelled in the LEGO proof-checker, originally written by Randy Pollack and maintained by the LEGO

group at Edinburgh University. I also made several further changes to the standard LEGO program in order to narrow the formalisation gap further; these are outlined in the next section. The resulting system is introduced in chapter 3, with one particular extension, that of coercion synthesis, described in chapter 4.

1.1.5 Literateness

The case-study is designed to be an prototype example of a *literate formalisation*. The term is derived from Donald Knuth's concept of literate programming, where a program is written in such a style that it can be read and understood by both a human reader and by the machine that will run the program. From a simple viewpoint, a literate formalisation is similar. The source files for the formalisation can be checked for correctness by a machine, but they are written in such a style so that when they are suitably processed they provide an account that is suitable for reading by a human mathematician.

Much of the literate processing for my case-study involves some simple pretty-printing of the development. LEGO developments are written in ASCII, which is hard to read compared to more conventional mathematical notation. I extended LEGO with some functionality so that it can output a pretty-printed account of the development it checks by following various directives and interleaving LaTeX commentary. The desire for literateness also affects the general writing style used, which is designed to allow the presentation of the material in question in a sensible order and with the emphasis on the information that is most useful to the reader. A few further extensions to standard LEGO help in this aim, although I believe far more work could be performed in this area. Some of this functionality is introduced along with the type theory in chapter 3, and the pretty-printing process is described in chapter 7.

More broadly, making a development in a formal system more literate means

to make its language more human-readable. This is clearly connected with a general narrowing of the formalisation gap.

One way in which this is done is through attempting to encode expressive informal mathematical idioms and use them in an unambiguous way. The main example of this in the case-study is through the extension of LEGO with a system for coercion synthesis; this is explained in detail within chapters 4 and 5. Another mechanism is through the explicit suppression of "uninteresting" material so as to present developments at a more appropriate level of detail. This methodology is discussed in chapter 6 and is illustrated by the main presentation of the case-study in chapter 10, which is an interweaving of informal explanation with formal expressions and machine-checked statements.

1.2 Objectives of the work

The main aim was to develop a formalisation of the fundamental theorem of Galois theory, described in chapter 2. Special attention was to be paid to the readability of the development, both on the smaller scale (expressions should take forms not too far from the phrasings and formulae used in informal texts) and also on the larger scale (the development should be well-structured, and be pitched at an appropriate level of detail.) At the start of the work, this was to be accomplished just through investigating and picking the best choice from a variety of definitional frameworks and proof styles. Later the aims expanded to include modifications to the proof-checker itself, with the addition of some new expression syntax and a literate environment with pretty-printing.

Some choices made and conclusions arrived at are relevant only for the LEGO proof-checker, some also for other type-theoretic frameworks, and some for formalisation in general. This should be bourne in mind when reading the thesis; I shall try to distinguish between the cases when it seems relevant.

1.3 Thesis outline

I provide summaries of the content of each of the chapters that follow. The structure of the thesis has been designed so that readers with different backgrounds or interests can find it a useful read without having to plough through all 243 pages, and so some alternative reading orders are also presented.

1.3.1 Chapter summaries

1. *Introduction.* You're reading it. Introduces the areas of interest, sets out some objectives, and outlines the structure of the remainder of the thesis.

2. *The mathematics to be formalised.* Provides an informal background to the mathematics that will be formalised in chapter 10. Contains a brief review of classical Galois theory, a discussion of how to make it constructive and easy to formalise, and a precise statement of the result together with an outline of the proof.

3. *The formal framework.* Introduces the formal framework, which is a variant of UTT implemented by a modified version of the LEGO proof-checker. All the relevant parts of the formal syntax are explained. Every following chapter relies on the reader having read this one.

4. *Coercions.* Introduces the concept of a coercion and describes the operation and implementation of a variant of LEGO extended with a system for synthesising implicit coercions. Examples of how to use the improved expressiveness found in the syntax of the theory that results are also given.

5. *Further coercions.* Explores some other ways in which coercions could be used and implemented and explains the design decisions taken in implementing LEGOwcs. Also considers the status of coercions relative to related concepts in type theory such as subtyping and overloading. The previous two chapters are partially based on material presented at TYPES'97.[Bai98].

6. *Proof style.* Looks at the decisions that have to be made about the overall style of proof one will use when working in a formal system. Practices useful in informal mathematics are identified and their possible translation to a formal system is discussed.

7. *Literate tools.* Presents the literate environment part of the extended LEGO system I developed and used in this thesis. Techniques of from literate programming and their relevance to and implementation in literate formalisation are discussed. The pretty-printing mechanisms coded are explained and an example of the written source from which the presentations found in this thesis are produced is given.

8. *Representation of foundational notions.* Reviews the foundations upon which the case-study formalisation is built. Choices in implementing the logical and set-theoretic basis for the development are explained. Sets, mappings, subsets and quotients are defined.

9. *Representation of common concepts.* Looks at the way in which some common mathematical concepts were formalised for the purposed of the case-study. These include isomorphism and function restriction, decideability and discreteness, the algebraic hierarchy of groups, fields and vectorspaces, finiteness of size and of dimension, and some interrelationships between these notions.

10. *The case-study development.* A long chapter presenting the case-study literate formalisation. The account is designed to be readable as a careful mathematical paper supported by a machine-checked formal proof.

11. *A short detailed proof.* As an addendum to the main presentation, which suppresses details in order to present the material at an appropriate level, one small part of the case-study is presented at a greater level of detail so that the reader can see what goes on at the level at which the formalisation is written.

12. *Related work and conclusions.* Reviews some other pieces of work in related areas. This review has been saved until the end as I wish to discuss the work with respect to material introduced throughout the thesis. The success or otherwise of the case-study is assessed and some suggestions of areas for further work are made.

A. *Resources.* Locates the electronic sources for the LEGO proof-checker and for the case-study development.

B. *Formalisation.* The entire formalisation in all its glorious and gory detail. This is contained in a separate volume in the printed version of this thesis.

1.3.2 Chapter dependencies

Figure 1.1 shows how the various chapters depend on each other. If one chapter depends on another, it rests upon it in the figure. If it is useful but not essential to have read some other chapter, then the first chapter is supported by struts originating in the chapter it is useful to have read. (Note that of chapter 6, only subsection 6.1.4.2 needs support from chapter 6.)

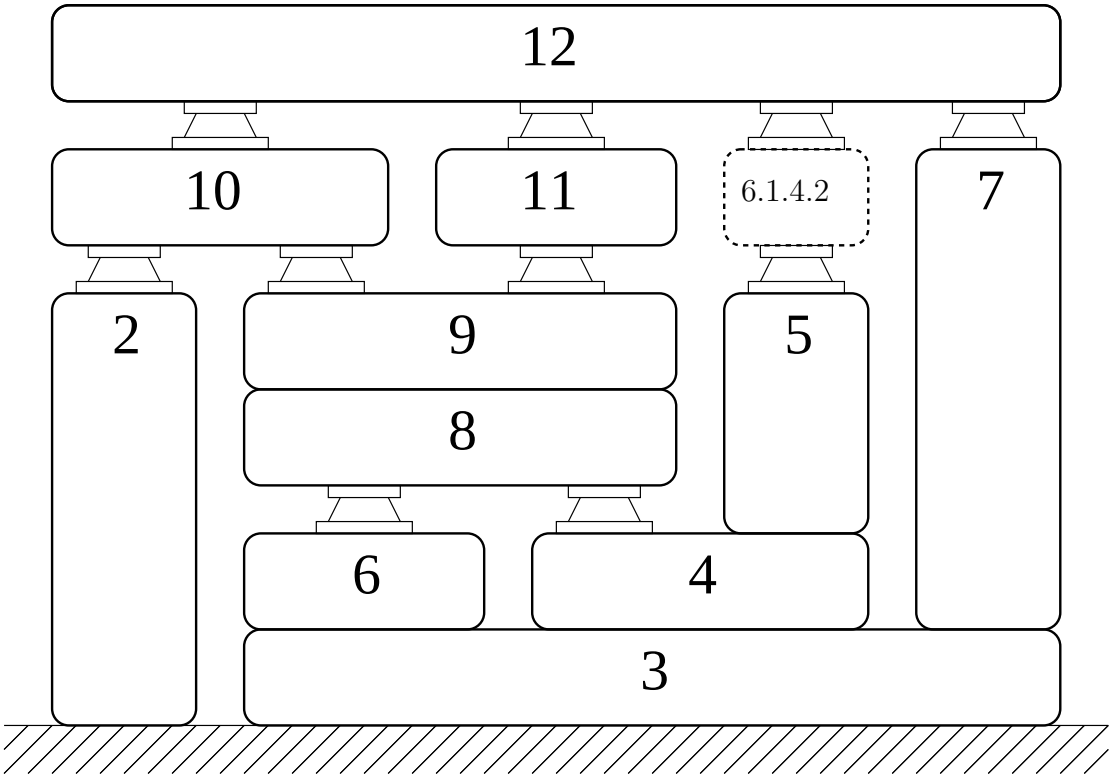

Figure 1.1: Dependencies between chapters

1.3.3 Ways to read the thesis

As well as reading this thesis straight through from beginning to end, you can also usefully read it only partially in the following four ways.

1. To get a brief overview: read chapter 3, browse the formalisations in chapters 10 and 11, and read the conclusions in chapter 12.

2. To read the mathematical case-study: first browse the introduction to the maths itself in chapter 2, read about the formal system in chapter 3, and browse information on coercions and the mathematical framework in chapters 4, 8, and 9. You should then be able to read all of the case-study presented in chapter 10, and may also wish to browse chapter 11.

3. To read about my work on coercion synthesis and its applications, read chapters 3, 4, and 5. You may also wish to check section 6.1.4.2.

4. To read the discussions of proof-style and find out about the literate environment I implemented, read chapters 3, 6, and 7.

These routes through the work are illustrated in figure 1.2 on page 27.

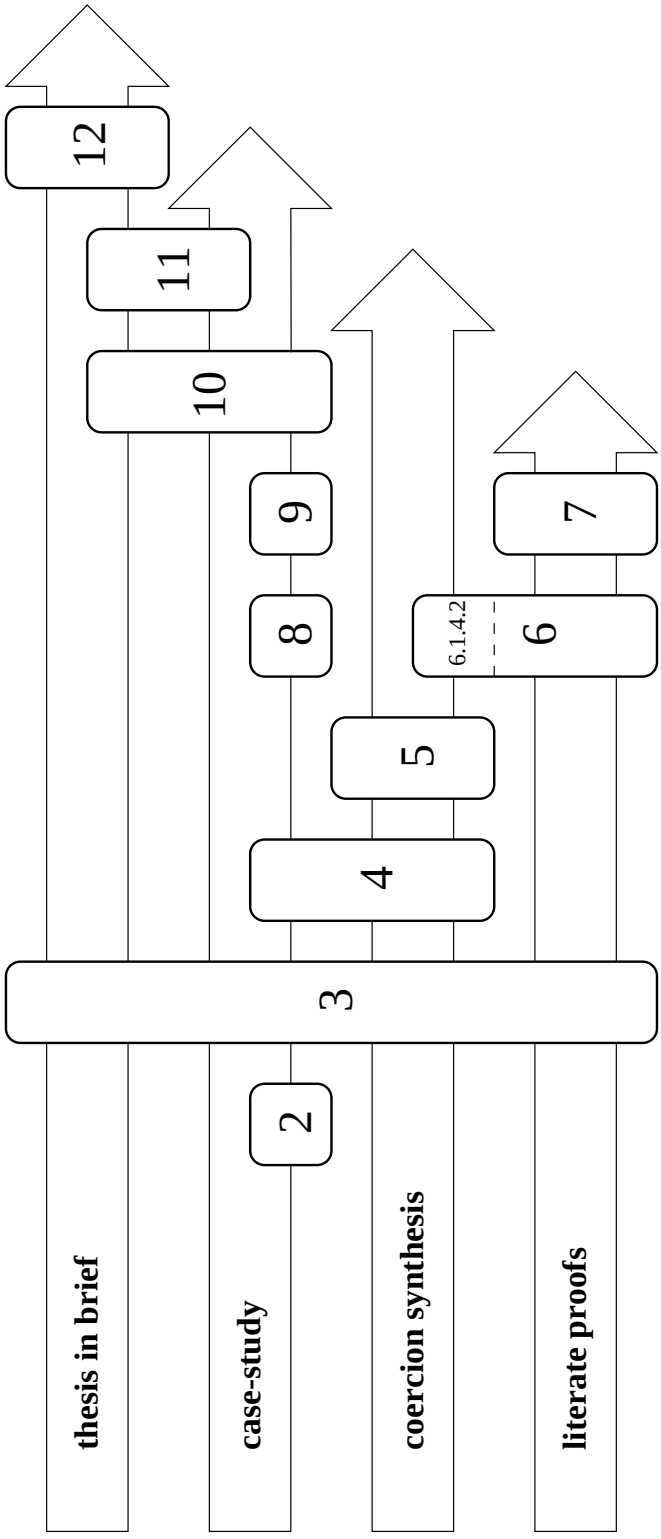

Figure 1.2: Four ways to read only parts of this thesis

Chapter 2

The mathematics to be formalised

This chapter provides an informal background to the mathematics that I will formalise later in the case-study. I briefly review a standard classical formulation of Galois theory and the context within which it is commonly used. The case-study will present a formulation of the fundamental theorem of Galois theory which is designed to be constructive and also easy to formalise; I comment on the differences between this formulation and the standard one. I then present a precise statement of the theorem, and outline the method of proof that will be used in the case-study, presenting some of the major subresults needed.

2.1 Classical Galois theory

2.1.1 Splitting field extensions

Galois theory[Ste79] is concerned with the behaviour of the roots of a polynomial $p \in K[X]$ in fields extending an original field of co-ordinates K. A field extending K is one of which K can be considered a subfield. Such an extension

field $L \supseteq K$ can also be considered as a vectorspace over K, and I will write this as $L : K$. I will be interested in the case when L can be generated by linear combinations (using scalars from K) of a finite tuple of elements $\boldsymbol{v} = (\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n) \in L^n$ that are algebraic over K. In this case I say that L is equal to the K-span of $\boldsymbol{v}$. The vectorspace $L : K$ then has a finite dimension no bigger than n; I write this dimension as $[L : K]$.

The polynomial p is said to split in L if it can be factorised as

$$p = x \prod_{i=1}^{n} (X - \boldsymbol{v}_i)$$

for a scalar $x \in K$ and roots $\boldsymbol{v} \in L^n$. It turns out that for every p there exists a *splitting field* L that splits p and that is the K-span of the roots of p, and that splitting fields for p are essentially unique in that any two such fields are related by a field isomorphism that has no effect on K. Because of this the behaviour of the roots of p in all possible field extensions of K can be studied by looking only at the case of any one particular splitting field.

2.1.2 The Galois connection

The fundamental theorem of Galois theory provides a way to study this behaviour through a Galois connection between field extensions and certain groups of automorphisms on those field extensions.

Given K and a p-splitting extension L, consider the set $\mathcal{F}(L, K)$ of fields J, for $K \subseteq J \subseteq L$, partially ordered by subset inclusion. Now, the field automorphisms of L form a group $\mathrm{Aut}(L)$. We can then also consider the set $\mathcal{G}(L, K)$ of the subgroups of this group containing only field automorphisms of L that are the identity on K. (The different automorphisms in such subgroups can be distinguished by the way in which they permute the roots of p.) This set $\mathcal{G}(L, K)$ is also partially ordered by subset inclusion.

Now define $(^\nabla) : \mathcal{G}(L,K) \to \mathcal{F}(L,K)$ and $(^\vartriangle) : \mathcal{F}(L,K) \to \mathcal{G}(L,K)$ by

$$\begin{aligned} G^\nabla &= \{x \in L \mid \forall g \in G.\ g(x) = x\} \\ J^\vartriangle &= \{g \in \mathrm{Aut}(L) \mid \forall x \in J.\ g(x) = x\} \end{aligned}$$

This pair of operators forms a Galois connection between the partially ordered sets $\mathcal{G}(L,K)$ and $\mathcal{F}(L,K)$. This can be axiomatised as

- If $G_1 \subseteq G_2$ then ${G_2}^\nabla \subseteq {G_1}^\nabla$,
- If $J_1 \subseteq J_2$ then ${J_2}^\vartriangle \subseteq {J_1}^\vartriangle$,
- $G \subseteq G^{\nabla\vartriangle}$, and
- $J \subseteq J^{\vartriangle\nabla}$.

The fundamental theorem of Galois theory will state that for a pair of certain collections of so-called Galois groups G and Galois subfields J, the inclusions in the last two axioms above can be replaced with equalities, so that the operators $(^\vartriangle)$ and $(^\nabla)$ form an anti-isomorphism between these groups and subfields. This is the main result that will be proved in the case-study; its application to polynomials and consequences (see the following subsection) have not been formalised therein.

2.1.3 The Abel-Ruffini theorem

One particular application of the fundamental theorem is to prove the Abel-Ruffini theorem. If K is the field of rational numbers $\mathbb{Q}$, then it is known that the roots of polynomials of low degree can be found using the field operators together with radicals (that is, root operations $\sqrt[i]{}$) on the coefficients of these polynomials. For example, the formula for the roots of quadratic polynomials, $x = (-b \pm \sqrt{b^2 - 4ac})/2a$, is recognisable to most high-school students; similar

formulas can be found for cubics and quartics. However, the Abel-Ruffini theorem states that when $n > 4$ then not all polynomials of degree n may be solved by such a method.

Whether or not a polynomial is solvable by radicals over K can be determined by examining the structure of $\mathcal{F}(L, K)$ for L a splitting field extension of K. In turn, the anti-isomorphism provided by the fundamental theorem of Galois theory allows us to express this structure in terms of a simple property of $\mathcal{G}(L, K)$, that of being *solvable*. It turns out that the symmetric group over a set with five or more elements does not have this property, from which the Abel-Ruffini result follows.

2.2 Differences from other presentations

The setting within which the case-study will be formalised varies from standard presentations in a few ways that are worthy of comment. Some of these are to make the formalisation process easier in general and more suited to a type-theoretic framework in particular. Some are also influenced by the preferences of Peter Aczel in determining the overall shape of the GALOIS project, of which the case-study formalisation is intended to form a substantial component.

In this section I review these differences and comment on their significance for the way in which the formalisation proceeds and the ways in which it might be used.

2.2.1 Constructivity

It is intended that the proofs used in the GALOIS project, and hence those presented in the case-study, should be constructive. One main reason for this is that the logic that naturally arises through the "propositions as types" paradigm

out of intuitionistic type theories, such as that used in LEGO, is constructive. Whilst one could assume the law of the excluded middle as an extra axiom and so carry through some of the classical proofs, there would seem to be little appeal in doing so when it is unnecessary; a constructive variant of the fundamental theorem of Galois theory is relatively simple to formulate, and its proof is not unattractive.

I should now discuss some of the extra things that must be taken into account in a constructive presentation of Galois theory. The first point to make is that all the fields considered must be discrete. (This means that equality in the field is decideable.) This will be the case for the fields which one wants to consider in the applications of the fundamental theorem, namely the rationals $\mathbb{Q}$ and algebraic and transcendental extensions to them, and so it can be safely assumed.

There are also two classically trivial but important results that need special attention. One is that the factorisation of a polynomial into irreducible factors is not constructive for all fields. However the factorisation will be possible for the rationals and its related extensions, and so again this problem can be safely ignored.

The second result concerns considering a field as a finite-dimensional vectorspace over one of its subfields. Classically, if $K \subseteq J \subseteq L$, and $L : K$ is finite-dimensional, then also $L : J$ and $J : K$ are finite-dimensional. This is not guaranteed constructively, and so I have to include these hypotheses about J in the statement of some of my results. Again, these facts turn out to be true for the fields of interest and so introducing these extra conditions should not cause problems in the future application of the case-study results.

2.2.2 Predicativity

Another feature desired for the GALOIS project is that its development should be predicative. An impredicative definition presupposes the existence of the object that is being defined, and refers to it in the definition. However, the definitions of the elements of a predicative type cannot, for example, themselves involve quantifications over this type.

An example of impredicativity that may be familiar to the reader is that of the standard definition of the set of natural numbers in set theory. Zero is defined, and so is a successor operator S. The set of natural numbers is then defined as

$$\mathbb{N} = \bigcup \{X \mid (0 \in X) \wedge (\forall n \in X,\ S(n) \in X)\}.$$

This definition is impredicative since $\mathbb{N}$ is itself one of the sets X over which the intersection is taken.

The type theory of LEGO does contain an impredicative type of propositions, but the case-study development does not make use of it. This means I cannot make use of results from the existing LEGO library, but as a result of this the development presented in this thesis is both constructive and predicative, in line with the desires for the GALOIS project.

2.2.3 The general proof method

Most presentations of Galois theory are not from first principles; instead they make use of results and constructions from existing bodies of mathematics, particularly algebra. In the case of the formalisation, such bodies of work were not already extant and therefore it was important that the method of proof used should stand on its own, and be direct and independent of other material. (There is a library of existing LEGO material, but it does not cover many of the results required, and is also based on the impredicative propositional type that I wished

to avoid using.)

The method of proof used in the case-study follows an informal but careful presentation written specially for the GALOIS project by Peter Aczel. This in turn followed some ideas presented in a paper[Dre95] by Andraes Dress; however, it was reworked in order to be constructive and also to be as independent as possible of other material. For example, polynomials do not feature in the development; the fundamental theorem is stated independently of this area of mathematics, though it can easily be applied to it. The formalisation itself is a reworking of Aczel's formulation taking into account concerns made relevant by the type-theoretic framework, one of which is described below.

2.2.4 Use of subobjects

In the experience of the author, formalisations using type theory tend to require that a greater distinction be drawn between subsets and sets (and between other related subobjects and objects) than is usual in traditional informal presentations.

Informal mathematics often takes place in a setting where one imagines some universal set within which all sets of interest are contained, and across which an intentional or "book" equality is defined. Hence we are used to taking intersections of arbitrary sets and considering inclusions between them. There is no great distinction drawn between sets and subsets.

Within most formulations of set theory within a type-theoretic framework, a set tends to provide an element type and a particular equality relation defined on that type. A subset is then defined relative to a particular set by means of a predicate over the element type. Intersections and inclusions between pairs of such subsets are easy to define because of the shared element type and equality relation. But the definitions are then not straightforward for sets, because different sets use different types and different equality relations.

This implementation distinction between sets and subsets is made invisible to some degree in the formalisation by means of coercions. Nonetheless, it is still present and needs to be paid attention to in order to make the formalisation as neat as possible, particularly when inclusions between objects are considered.

The standard presentation of Galois theory starts with a field which is then extended by one of a collection of larger fields that contain it. My formalisation instead considers the original field K and its possible extensions as (decideable) subfields of a larger universal field F. In the intended areas of application, F will be some algebraic and transcendental extension of the rational numbers, which will themselves comprise the subfield K.

Similarly the automorphisms of K and other fields are not defined as independent groups in their own right but as subgroups of one universal group of permutations of the whole of F. A permutation of a subfield is defined as a permutation of F that is the identity outside of the subfield in question. This formulation is adequate, since the subfields in question are decideable and hence the domain of a permutation of a subfield can be extended to cover the whole of F without falling foul of constructivity requirements.

2.3 Statement of the fundamental theorem

As previously explained, I work in a discrete universal field F. It has a group of permutations, perm(F). I will consider subfields $L \subseteq F$; the permutations of L will be the subgroup $\mathrm{Perm}(L) \subseteq \mathrm{perm}(F)$ of functions that are the identity outside of L. A further qualification produces another subgroup $\mathrm{Aut}(L) \subseteq \mathrm{Perm}(L)$; these are the field automorphisms of L, permutations that respect addition and multiplication in L.

The Galois connection operators are now defined relative to L. For $G \subseteq$

$\mathrm{Aut}(L)$, $\mathrm{fix}(L, G)$ (also written $G^{\triangledown}$) is the subfield of L fixed by every automorphism in G, and for $K \subseteq L$, $\mathrm{aut}(L, K)$ (also written $K^{\triangle}$) is the subgroup of $\mathrm{Aut}(L)$ containing the automorphisms that fix everything in K.

A *Galois group* is now defined to be a finite subgroup $G \subseteq \mathrm{Aut}(L)$ such that $L : G^{\triangledown}$ is finite-dimensional. A *Galois subfield* is defined to be any $K \subseteq L$ that is equal to the fixed subfield $G^{\triangledown}$ of some Galois group G.

Finally given any $K \subseteq L$, I define a *K-subfield of L* to be any decideable subfield J with $K \subseteq J \subseteq L$ and with both $L : J$ and $J : K$ finite-dimensional.

With the setting completed, I can now state the fundamental theorem. It divides into three thirds, each of which has several parts. Throughout, let $L \subseteq F$ be decideable.

(1) Let G be a Galois group of L.

- An auxilliary result: suppose U is a decideable subgroup of G with $L : U^{\triangledown}$ finite-dimensional, then U is also a Galois group of L.
- $G^{\triangledown}$ is a Galois subfield of L, and its fixing subgroup $G^{\triangledown\triangle} = G$.

(2) Now let K be a Galois subfield of L.

- Conversely to part 1, let J be a K-subfield of L. Then J is also a Galois subfield of L.
- $K^{\triangle}$ is a Galois group of L, and its fixed subfield $K^{\triangle\triangledown} = K$.

The anti-isomorphism between the Galois groups and subfields of L induced by the operators $(\triangledown)$ and $(\triangle)$ is thus established.

(3) The final third of the fundamental theorem relates the Galois subfields of different subfields. As in part 2, let K be a galois subfield of L, and J a K-subfield. Then K is a galois subfield of J if and only if $J^{\triangle}$ is a normal

subgroup of $K^{\triangle}$. Also, when this is the case, then the quotient group $K^{\triangle}/J^{\triangle}$ is isomorphic to $\mathrm{aut}(J, K)$.

2.4 Outline of the proof method

There are four major subresults that are used in the proof of the fundamental theorem and also in the proof of each other.

2.4.1 Lemma 1

For a subgroup $G \subseteq \mathrm{Aut}(L)$ and a decideable subfield $J \subseteq L$, define $U = G \cap J^{\triangle}$ and let $G \restriction J = \{g \restriction J \mid g \in G\}$, the restriction of the automorphisms in G to J.

Lemma 1 states:

(i) For $g_1, g_2 \in G$, the cosets g_1U and g_2U are equal if and only if $g_1 \restriction J = g_2 \restriction J$. This is proved by straightforward equational reasoning

(ii) $G/U \simeq G \restriction J$. This follows from the first result by defining an isomorphism $\theta(g) = g \restriction J$.

2.4.2 Theorem A

Suppose G is a Galois group on L and let $K = G^{\triangledown}$.

Theorem A states:

(1) If J is a K-subfield of L, then the set $\mathrm{FixHom}(J, K)$ of field homomorphisms from J into L that fix K has a size smaller than $[J : K]$.

(2) If the subgroup $U \subseteq G$ is such that $U^{\triangledown} : K$ is finite-dimensional, then the quotient G/U has size equal to $[U^{\triangledown} : K]$.

This is proved by using Dedekind's lemma (a result concerning the linear independence of certain mappings) in combination with a well-known fact about group

actions. This is the part of the development that was not formally proved, due to a lack of time. (I did also formalise Dedekind's lemma separately as an early case-study, but that work is not presented in the thesis.)

A useful corollary to the second result is:

(3) G has size equal to $[L : K]$.

This follows by setting U equal to the trivial subgroup of G that contains only the identity.

2.4.3 Theorem B

Again let G be a Galois group on L, write K for G^{∇}, and let J be a K-subfield of L.

Theorem B states:

(0) $G \upharpoonright J$ has finite size

(1) $G \upharpoonright J = \mathrm{FixHom}(J, K)$

(2) $G \upharpoonright J$ has size equal to $[J : K]$

(3) $J = J^{\Delta\nabla}$

Let $U = G \cup J^{\nabla}$. The proof then uses Lemma 1 and Theorem A. It works by considering the size of G/U, $G \upharpoonright J$, and $\mathrm{FixHom}(J, K)$, and the dimensions $[J : K]$ and $[U^{\nabla} : K]$. These are shown to be related by a cycle of inequalities where each term is no larger than the next. This shows all are in fact equal, from which the required results can be derived. It also follows that

(4) $G \upharpoonright L = \mathrm{FixHom}(L, K)$

by setting J equal to L.

2.4.4 Lemma 2

The first two thirds of the fundamental theorem can now be proved, in order, by using Theorem B. In turn, all these results can be combined to prove a final lemma. Suppose K is a Galois subfield of L, let $G = K^{\Delta}$ and J be a K-subfield of L.

Then Lemma 2 states:

(i) If $J^{\Delta} \subseteq G$ is normal, then $G \restriction J$ is equal to the subset of mappings $\mathrm{aut}(J, K)$.

(ii) Conversely, if $G \restriction J$ and $\mathrm{aut}(J, K)$ have the same size, then $J^{\Delta} \subseteq G$ is a normal subgroup.

Combining all four subresults with the two thirds of the fundamental theorem already proved then allows me to complete the proof of the final third of the fundamental theorem. The group isomorphism used is the restriction isomorphism θ from Lemma 1.

Chapter 3

The formal framework

This chapter introduces the formal framework with respect to which this thesis will be written. This framework is a version of the type theory UTT as implemented by an experimental variant of the LEGO proof-checker. This variant offers some extensions to the allowable syntax for terms, and also generates a pretty-printed account of what it checks. The system will be presented in this chapter and used in the thesis as a composite package, but for the benefit of the reader already used to LEGO, I will identify which parts of this package involve the extended features. The main changes an existing user of LEGO will see in this chapter will be in the style of presentation, which avoids the ASCII notations of traditional LEGO. Further details of the pretty-printing process are discussed in chapter 7. Where to find LEGO and my additions to it is explained in appendix A.

3.1 The Unified Theory of Types, UTT

The Unified Theory of Types (henceforth known as UTT) is an extension of Zhaohui Luo's extended calculus of constructions (ECC) with inductive types and rewriting. The ECC[Luo90] is itself an extension of Coquand and Huet's

original calculus of constructions[CH88] with strong sum types, an impredicative type of propositions and a hierarchy of type universes with subsumption.

I shall build up these layers and introduce the different constructors of UTT by means of examples. I refrain from giving a full and formal presentation of the judgement rules of UTT since this is unlikely to be revealing to those new to the theory and will be familiar ground to the more experienced reader. A full such presentation of the type theory can be found in the paper[Luo94] in which Luo introduces UTT.

3.1.1 Basics

The calculus of constructions can be seen as a higher-order λ-calculus. There are various formation rules for terms of the calculus. Terms can always be typed; their types are also terms. I write the judgment that a term x has type A as $x : A$. In order to state a judgement that has been checked by the formal system, I write it on a separate line like this:

$\triangleright \quad x : A$

Any term that can appear on the right-hand side of a typing judgement is called a *kind* in the terminology of this thesis. For this introductory subsection, I shall consider a single type of kinds, called "Type". In subsection 3.1.3 I shall introduce a full hierarchy of type universes and explain their relationship.

If $A : \mathsf{Type}$, and, if given that $x : A$ then also $B : \mathsf{Type}$, then we may form a dependent product type, or Π-type.

$\triangleright \quad \{\Pi x : A\}\ B : \mathsf{Type}$

If the term x does not occur in B, then I call this a non-dependent product, and may write it in an alternative form that does not name x:

$\triangleright \quad A \to B : \mathsf{Type}$

Suppose that for $x : A$, the expression e has type B. Then we may form a

λ-abstraction of x over A:

▷ $[\lambda x : A]\ e : \{\Pi x : A\}\ B$

Given a term $f : \{\Pi x : A\}\ B$ (which we might call a function), and a second term $a : A$ (which we might call an argument), I can form the application of the function to the argument,

▷ $f\ a : B[a/x]$

where $B[a/x]$ is B with a substituted for x.

The application of a λ-term to an argument can be β-reduced in the way standard in a λ-calculus, by substituting the argument for the abstracted variable in the body of the abstraction.

If one term may be reduced to another, the terms are said to be *convertible*. Convertibility is an equivalence relation over terms, and one may think of convertible terms to have the same computational meaning, although they may be syntactically distinct. I use the symbol $\simeq$ to denote a convertibility relationship between two terms that has been checked by the formal system. Hence the β-conversion of a λ-term application may be written:

▷ $[\lambda x : A]\ e\ a : B[a/x]$

▷ $[\lambda x : A]\ e\ a \simeq e[a/x]$

In practice, conversion judgements are used to demonstrate computations, and also to summarise the important information about material the details of which have been omitted. Because of this, sometimes they may be read as if they are definitions of the identifiers whose properties are being summarised. This variety of use is discussed further in chapter 6 on page 136.

3.1.2 Strong sum types

As an analogue to dependent products, UTT also features dependent sum types, or Σ-types.

▷ $\langle \Sigma x : A \rangle\ B$ **:** Type

These are types of terms that are pairs with components in types A and B.

▷ Introduce b **:** $B[a/x]$

▷ $(a, b : \langle \Sigma x : A \rangle\ B)$ **:** $\langle \Sigma x : A \rangle\ B$

When B does not depend on x (and hence $B[a/x]$ is the same as B), I write the non-dependent type in an alternative simpler form, just as for $\{\Pi x : A\}\ B$ and $A \to B$.

▷ (a, b) **:** $A \# B$

A pair of terms that can be given a dependent Σ-type also has a simpler type that does make the dependency explicit. This means that the explicitly dependent pair $(a, b : \langle \Sigma x : A \rangle\ B)$ could also be considered as a pair of terms with non-dependent types:

▷ (a, b) **:** $A \# B[a/x]$

The two types $\langle \Sigma x : A \rangle\ B$ and $A \# B[a/x]$ are not convertible, and so in fact there are two different flavours of the pairing constructor, a non-dependent and a dependent pairing. The non-dependent type is the default. Dependent pairings specify the dependent type within the term itself as seen in the previous paragraph.

If p is a pair, then the two individual components of the pair are written as the projections $p.\mathbf{1}$ and $p.\mathbf{2}$.

▷ $(a, b).\mathbf{1} \simeq a$

▷ $(a, b).\mathbf{2} \simeq b$

3.1.3 Universe hierarchy

One of the features of UTT is an impredicative type of propositions, Prop. It is impredicative because Π-types using abstraction over Prop are themselves contained in Prop.

▷ $\{\Pi P : \mathsf{Prop}\}\ P \to P : \mathsf{Prop}$

However, I avoid the use of impredicative constructions in this thesis. I mention the Prop type only because it is part of standard UTT and so it is present in the LEGO implementation of this type theory; I will make no use of it and the reader may safely forget about it.

In subsection 3.1.3 I worked with a single type of kinds, Type, for the sake of a simpler explanation. In fact UTT contains a sequence of type universes, Type_0, Type_1, Type_2, ... In general we have that

▷ $\mathsf{Type}_i : \mathsf{Type}_{i+1}$

and so on. The universes are also *subsumptive* in that each one contains everything that it's predecessor did; if the types are considered as sets we might write that $\mathsf{Type}_i \subset \mathsf{Type}_{i+1}$. In the type theory this translates to a rule that if

▷ $A : \mathsf{Type}_i$

we also have that

▷ $A : \mathsf{Type}_{i+1}$

and so on.

This sequence of progressively higher type universes give one a predicative way to give a type to a term that itself quantifies over all of a type universe. So long as the term B can be typed as Type_{i+1}, we have that

▷ $\{\Pi x : \mathsf{Type}_i\}\ B : \mathsf{Type}_{i+1}$

▷ $\langle \Sigma x : \mathsf{Type}_i \rangle\ B : \mathsf{Type}_{i+1}$

I shall not need terms outside of Type_2 in the case-study presented in this thesis.

3.1.4 Inductive types

UTT offers the ability to define the terms of certain types and constructions over these types inductively. The introduction and elimination operators are

appropriately typed objects together with some special rewrite reductions that ensure certain pairs of expressions are convertible.

A standard example of an inductive type is that of the natural numbers.

▷ $\mathsf{nat} : \mathsf{Type}_0$

▷ $\mathsf{zero} : \mathsf{nat}$

▷ $\mathsf{succ} : \mathsf{nat} \to \mathsf{nat}$

The details of elimination and rewriting tend to look complicated because they need to generalise all possible constructions over the inductive type. Some insight into how they work can be gained by noting that if C is considered to be a predicate, then $\mathsf{nat\_elim}\ C$ proves C holds for all natural numbers by induction.

▷ $\mathsf{nat\_elim} : \{\Pi C : \mathsf{nat} \to \mathsf{Type}_1\}\ (C\ \mathsf{zero}) \to$
$(\{\Pi n : \mathsf{nat}\}\ (C\ n) \to C\ (\mathsf{succ}\ n)) \to \{\Pi n : \mathsf{nat}\}\ C\ n$

▷ Reductions: $[\lambda C : \mathsf{nat} \to \mathsf{Type}_1]\ [\lambda n : \mathsf{nat}]\ [\lambda \mathit{zero\_case} : C\ \mathsf{zero}]$
$[\lambda \mathit{succ\_case} : \{\Pi n' : \mathsf{nat}\}\ (C\ n') \to C\ (\mathsf{succ}\ n')]$
$\mathsf{nat\_elim}\ C\ \mathit{zero\_case}\ \mathit{succ\_case}\ \mathsf{zero} \Longrightarrow \mathit{zero\_case}$
$\|\ \mathsf{nat\_elim}\ C\ \mathit{zero\_case}\ \mathit{succ\_case}\ (\mathsf{succ}\ n) \Longrightarrow$
$\mathit{succ\_case}\ n\ (\mathsf{nat\_elim}\ C\ \mathit{zero\_case}\ \mathit{succ\_case}\ n)$

In general in presenting material I will not make these elimination operators explicit and instead directly present individual constructions that have been defined in terms of them. For example, suppose double represents the following application of nat_elim:

▷ $\mathsf{nat\_elim}\ ([\lambda \_ : \mathsf{nat}]\ \mathsf{nat})\ \mathsf{zero}\ ([\lambda \_, n : \mathsf{nat}]\ \mathsf{succ}\ (\mathsf{succ}\ n)) : \mathsf{nat} \to \mathsf{nat}$

Then the rewrite rules applied to double give the following conversions, which are somewhat easier to read and may be thought of as defining the operation of the function:

▷ $\mathsf{double}\ \mathsf{zero} \simeq \mathsf{zero}$

▷ $\mathsf{double}\ (\mathsf{succ}\ n) \simeq \mathsf{succ}\ (\mathsf{succ}\ (\mathsf{double}\ n))$

An example of this form of presentation can be found on page 190.

3.1.5 Relation to Martin-Löf type theory

One of the desires for the GALOIS project, of which my case-study is designed to be a part, is that it should follow the school of constructivism advocated by the Göteborg group[NPS90]. Within this approach all terms of the formal theory have *meaning explanations* given to them through computations, which is seen as a desirable thing. Currently this necessitates that the formal system involved be predicative. Users of impredicative systems also tend to encode a logic[Jon96] that is intuitionistic, but not truly constructive as far as disjunction and existential quantification are involved.

Ideally GALOIS would use a machine implementation of a variant of Martin-Löf type theory with suitable extensions such as universes and inductive types. However it was currently more convenient for me here at Manchester to use UTT within the LEGO proof-checker. Although UTT is more powerful than Martin-Löf type theory, the case-study does not rely on any of these more powerful features in any fundamental way, and so a reformulation of the UTT formalisation should not present any problems. In particular, the case-study avoids impredicativity altogether, and does not use the impredicative and partially non-constructive encoding of logic that is the UTT/LEGO standard.

3.2 LEGO extensions to UTT

The LEGO proof-checker offers an implementation of the type theory UTT (and also some other type theory variants with which we are not concerned.) As well as offering lots of extra functionality for managing a context of defined and declared terms, it also offers some small extensions to the type theory itself and allows some lexical abbreviations in writing terms of the theory. The non-standard version of LEGO used in this thesis includes some further moves in this direction.

3.2.1 Principal types and type-casting

Every LEGO term has many different types. In the first place, the type theory includes rules so that if $x : A$ and $A \simeq A'$, then x also has type A'. But terms also have different types which are not convertible, for example due to type universe subsumption.

When the LEGO system type-checks a term, it derives a so-called "principal type" for it. The type derived for a term will be built in a canonical way from the principal types of its subterms, and LEGO will try to use the lowest type universes that it can. By default, LEGO types an otherwise untyped term with this principal type.

However, sometimes one will wish to associate a different principal type with some term. This could be just for the display purposes, or it might have some later effect on the operation of the system. (For example, using a higher level in the type universe hierarchy. The coercion synthesis system introduced later in the thesis will also give a special role to the principal types of terms.)

One can do this by supplying the term that one would like the term to have as its principal type explicitly. This is known as *type-casting*. I use type-castings intermittently in the thesis for the sort of reasons mentioned in the previous paragraph. I don't make the casting explicit within the term itself but it can be observed in the type that it is given.

3.2.2 Local definitions within terms

LEGO allows a few tricks of syntax to allow terms to be expressed in more succinct ways. One of these is a mechanism of local definitions for the sake of abbreviation. Suppose that the expression e contains multiple instances of some long_expression. Then these instances can be abbreviated by some identifier x within the term e by writing

▷ $[\delta x = \mathsf{long\_expression}]\ e[x/\mathsf{long\_expression}] \simeq e$

3.2.3 Argument synthesis

A second syntax trick that LEGO allows is the automatic synthesis of certain function arguments. Since the types of later arguments to a function may depend on the values of previous ones, in certain circumstances sensible values for the previous arguments can be inferred from the later ones. This means that the previous ones need not be supplied explicitly. They could be left implicit and synthesised from the context in which the function is used.

To prevent ambiguities, whether an argument to a function should be left implicit is determined by the flavour of binding used in the relevant abstraction within the function. An *explicit* binding uses the colon symbol that you have already seen, ":". An *implicit* binding uses the vertical bar symbol, "|", instead. The different flavours of binding are considered convertible.

▷ $\{\Pi x \,|\, A\}\ B \simeq \{\Pi x : A\}\ B$

▷ $[\lambda x \,|\, A]\ e \simeq [\lambda x : A]\ e$

As an example I take the identity function. This function may be considered to be "polymorphic" in that it could be applied to arguments of many different types. I first consider a version of this function that does not use an implicit binding to allow argument synthesis.

▷ $[\lambda T : \mathsf{Type}_0]\ [\lambda t : T]\ t \mathbf{:} \{\Pi T : \mathsf{Type}\}\ T \to T$

I'll abbreviate this λ-term $[\lambda T : \mathsf{Type}_0]\ [\lambda t : T]\ t$ as ID in the following.

The function ID could be applied to $a : A$ or to $b : B$ by supplying the types A or B as an initial argument.

▷ $\mathsf{ID}\ A\ a \mathbf{:} A$

▷ $\mathsf{ID}\ B\ b \simeq b$

Since LEGO can compute the principal types A and B of a and B, these type

arguments are redundant information. The clarity of expressions benefits from the elimination of such redundancies. A better formulation of the identity function might thus be to make the first variable binding an implicit one:

▷ $[\lambda T \,|\, \mathsf{Type}_0]\ [\lambda t : T]\ t : \{\Pi T \,|\, \mathsf{Type}\}\ T \to T$

Such a version of ID can be applied directly to a or to b and the implicit arguments A and B will be synthesised by the proof-checker.

▷ $\mathsf{ID}\ a \simeq a$

▷ $\mathsf{ID}\ b : B$

If one wants to explicitly supply an argument that has been specified as implicit, a different syntax for the application is used:

▷ $\mathsf{ID}|A\ a : A$

This can be useful on the occasions when one wants to partially apply a function by not supplying a later argument that would normally be supplied:

▷ $\mathsf{ID}|B \simeq [\lambda t : B]\ t$

▷ $\mathsf{ID}|B : B \to B$

3.2.4 Coercion synthesis

Argument synthesis works by the LEGO system automatically calculating a missing "obvious" argument to a function in order to make a term type-check. The version of LEGO used in this case-study contains a further mechanism of coercion synthesis, which automatically calculates a missing "obvious" coercion function that will be applied to an argument to achieve the same goal. The mechanism is more complicated and will be discussed in detail in chapters 4 and 5, but the aim is the same as for argument synthesis: to allow information in a term that is redundant to be omitted.

3.3 LEGO context and functionality

3.3.1 The LEGO context

A LEGO development is carried out relative to a context of definitions and declarations of terms. The context is a linear list, in which later entries can make use of preceding ones. Entries in the context may be *definitions* or *declarations*, *global* or *local*, and *unfrozen* or *frozen*.

3.3.1.1 Definitions

Definitions are introduced into the context by providing an identifier, and a term that will be represented by the identifier. One may see definitions as abbreviations that the LEGO system can expand.

▷ Define $X =$ long_expression $: A$

▷ $X \simeq$ long_expression

Definitions are global by default. A local definition is introduced like this:

▷ Let $Y =$ long_expression $: A$

Local definitions may be *discharged* from the context. When this happens, they become local definitions within the body of each individual term that references them.

▷ Define $Z = f\ Y : B$

▷ Discharge Y

▷ $Z \simeq [\delta Y =$ long_expression$]\ f\ Y$

Definitions may also be frozen. A frozen identifier cannot be expanded to become its definiens. LEGO allows identifiers to be frozen and unfrozen at any time, but in this thesis the frozenness of an identifier will be fixed at the moment of definition.

▷ Define frozen $Y = X : A$

Because Y is frozen above, it is not considered convertible with X even though that was its definiens.

Entries can be removed from the context by *forgetting* them and subsequent entries. There is no need to forget entries in the course of a development, but occasionally I may want to indicate that entries introduced only as examples are being forgotten.

▷ Forget back through X

I will use forget without formally noting it for the remainder of this chapter.

3.3.1.2 Declarations

The other sort of entry that can be introduced into the context is a declaration. A declaration is an identifier that is introduced and declared to have a certain type, but it is like a frozen definition in that its content is unknown, and so it is not convertible to some other particular term.

▷ Introduce $X : A$

Declarations are local by default. A global declaration may be introduced like this:

▷ Globally declare $Y : A$

Like local definitions, local declarations may be discharged from the context. When this happens, terms which depend on the discharged identifiers are generalised by prefixing them with an appropriate abstraction.

▷ Let $Z = f\ X : B$

▷ Discharge X

▷ $Z : \{\Pi X : A\}\ B$

▷ $Z \simeq [\lambda X : A]\ f\ X$

In normal LEGO, local definitions and declarations are discharged simultaneously, but the version I use in this thesis allows me to discharge local declarations whilst

leaving local definitions in the context, as was done above.

Another facility is the ability to *discharge and keep* some local entries. This is simply a shorthand for performing a discharge of those entries and then reintroducing them back into the context. It is used to generalise later entries over the local ones without losing those local ones from the context.

3.3.2 Types as propositions

Type theories are used as a framework within which logical proofs can be formalised by using the Curry-Howard correspondence between propositions and types. A proposition is represented by a type, and terms with that type then constitute proofs of the proposition. This gives rise to a constructive logic whose operators either correspond to existing type-theoretic operators of UTT, or can easily be defined in terms of them. The details are explained in chapter 8, but a simple example is that of logical implication. If the types P and Q are propositions, then the proposition that P implies Q is the functional type $P \to Q$. Terms with that type take proofs of P and convert them into proofs of Q; this is precisely what a constructive proof that $P \to Q$ should do.

In the above example, the symbol "$\to$" reads nicely both as a function arrow and as the implication operator. Some other operators of UTT that correspond directly to logical operators are Π- and Σ-abstraction: these are used to represent universal and existential quantification. I allow the use of the more familiar notation for quantification when this is the case. Thus if p is a predicate over some type T I write

- $\{\forall x : T\}\ p\ x \simeq \{\Pi x : T\}\ p\ x$
- $\langle \exists x : T \rangle\ p\ x \simeq \langle \Sigma x : T \rangle\ p\ x$

How the sort of use is specified within LEGO is explained in chapter 7. The reader need know only that the above notations are treated as equivalent by the

type-checker and differently only by the pretty-printer.

Context entries that have propositional types are also presented differently by the system although again the difference is purely lexical and the entries are treated in the same way by the proof-checking component of LEGO. The introduction of a global declaration becomes

▷ Assume without proof $X : P$

whereas the introduction of a hypothesis for later discharge as a local declaration is written

▷ Suppose $Y : P$

A definition with a propositional type is taken to be a proof of that proposition. Once we know a proposition has been proved we are rarely concerned with the details of that proof, and so proofs are usually frozen; I take this as a default and comment only if they are non-frozen. Also I write the proposition that is being proved first, and use a smaller typeface for the actual term that inhabits this type and constitutes a proof of it.

Thus a global proof definition is written

▷ Prove $X : P$
$= \mathsf{long\_expression}$

rather than the normal

▷ Define frozen $X' = \mathsf{long\_expression} : P$

and a local proof definition that will later be discharged after it has been used is presented as

▷ Prove subresult $Y : P$
$= \mathsf{long\_expression}$

3.3.3 Construction by refinement

LEGO offers the user the ability to construct proofs and other terms by *refinement*. This is done by stating a goal and then breaking this goal down into

subgoals by means of interactive proof commands. Finished refinement proofs are thus a sequence of proof commands that are performed in sequence by LEGO in order to generate a proof-term. This is a convenient way of working for the person doing the proof as they can allow LEGO to automatically fill in some of the details from the context of the subgoals currently being worked upon.

However, such top-down refinement proofs are very hard to read once they are written because the reader lacks the subgoal context that LEGO and the writer had when the proof was interactively constructed. Thus I do not present such sequences of proof commands in the thesis. Instead I tend to work by a process of incremental definition from the bottom-up that makes this context clear.

Combining bottom-up and top-down approaches is common in informal mathematics, and I do attempt this to some extent in the case-study. Further discussion of this and of the related issue of the difference between procedural and declarative proof-styles follows in chapter 6, and some functionality to allow goals to be stated in advance is presented in subsection 3.3.6.

Refinement proofs can still be useful in my presentation to discharge a relatively small step of a larger proof. Suppose I have a step that a human reader is happy to accept but that LEGO may prefer to be broken down into a few smaller substeps. I can then set this step up as a goal and construct the proof by refinement. In the presentation, this would be presented as

$\triangleright$ Construct by refinement $X : P$
(long_expression, P, ID)

Here ID, P and long_expression are various of the subterms that happened to appear at least once in the (nonsensical example) proof-term that was constructed.

In this way the reader can quickly see from which subresult or subresults the step followed, without having to parse the full details of the proof-term. Some rather more concrete examples of this sort of approach are found in the case-study proofs presented in chapter 10.

3.3.4 Fixity

LEGO allows any function application to be written postfix by using an alternative "dot" application operator.

▷ $a.f \simeq f\ a$

For the extended version of LEGO used in this thesis, I allowed functions specifically declared to be postfix to suppress the dot operator, and also allowed functions to be declared to have some other common fixities (such as infix.) Details of these fixities and declarations can be found in chapter 7.

3.3.5 Default types

Another lexical extension I implemented was the ability to define a default type for an identifier. This means that when the type of an introduced or abstracted identifier is the "obvious" one, it may be suppressed. Thus if x is declared to have the default type A, then I can write

▷ $[\lambda x]\ e \simeq [\lambda x : A]\ e$

I do not make much use of this facility in the case-study presented in chapter 10, because the material that declares many default types previously will not have been seen by the reader and so the omitting of the typing information would not aid the presentation. However I do make more use of it when presenting the entirety of the case-study in the appendix.

3.3.6 Long-term goals

In most mathematical presentations the writer will state the results and major subresults they hope to prove near to the start, to give a picture of the structure of the proof and motivate the development. LEGO allows one to state a goal and work towards it by refinement, but once start a refinement begins it must be finished before other work can proceed: it lacks a good context management

system that allows one to work on a combination of partially proved results. (LEGO provides a user with the more general ability to "cut" all instances of a declared term with a definition, but in practice this feature is too inefficient to use in proofs of any size since all later context entries have to be rechecked.) Since the ability to state a goal ahead of time seems important in producing readable presentations, I therefore extended LEGO with some basic functionality along these lines.

At any point in a presentation, I may state an intention to prove some particular (named) result:

▷ We want to prove $Y : P$

Later on, having incrementally built up some

▷ long_expression : P

I can use this as a proof-term to discharge the proof obligation:

▷ Prove as claimed $Y : P$
= long_expression

When I do this LEGO type-checks the definition in the standard way, but also checks that the type that long_expression has matches the one that earlier I had stated X should have.

I can also combine the discharge of local declarations with these long-term goal statements to a limited extent. Having declared

▷ Introduce $t : T$

▷ We want to prove $Z : p\ t$

then after I discharge t,

▷ Discharge t

the type required for Z will be $\{\forall t : T\}\ p\ t$ as one might expect.

Discharging can also interact with the fulfillment of the proof obligation.

▷ Introduce $t' : T$

If I now construct a proof for this t' I have introduced,

▷ the_proof : $p\ t'$

I can then discharge t' to generalise this proof so that it can be used to fulfill the proof obligation Z:

▷ Discharge to prove as claimed Z : $\{\forall t : T\}\ p\ t$

using

the_proof : $p\ t'$

▷ Discharge t'

Many examples of uses of these experimental functionality can be found in the case-study within chapter 10. Additional notes on the implementation of the mechanisms explained in the last three subsections and the reasons I found these extensions desirable can be found in chapter 7.

Chapter 4

Coercions

A coercion is a function that is applied to an object in order to give it a different type. The idea is that this allows us to use one object to represent some other related one; the coercion encodes the relationship between this pair of objects.

Coercions can be used to give a computational meaning to subtyping, and also to some other similar idioms; for example, a coercion could also translate between two different representations of an object. Alternatively, it can provide a new interpretation of an object altogether; in this manner, it can give a similarly computational meaning explanation to more general sorts of overloading and abbreviation.

This chapter describes how I have extended the standard LEGO proof-checker with a system for the automatic synthesis of coercions. I start by giving the background for the development of this system, and motivate by illustrating the role that coercions may be considered to play in the formalisation of common mathematical idioms. I proceed to describe the notions of an implicit syntax, and coercion synthesis. A precise description of the operation of the coercion synthesis system and some details of how it was implemented then follow, together with some simple examples of how it can be used.

4.1 Introduction

4.1.1 History

My own work on coercions is a development of previous research by Gilles Barthes and Peter Aczel. It took some inspiration from an implementation of classes in LEGO started by Randy Pollack as a result of their work, and also benefited greatly from collaboration with Amokrane Saïbi, who helped with the first implementation of coercions in LEGO, and went on to implement coercions in the Coq proof-checker.

Aczel and Barthes started looking at notions of class and inheritance between classes (as seen in object-oriented programming) whilst working on the GALOIS project. Since Galois theory draws together many pieces of abstract algebra, it involves a dense hierarchy of algebraic objects such as sets, groups, fields, and vectorspaces. There are various functions that can plausibly act on more than one of these types of object; these include operators, theories, proofs, and the construction of other structures. In the object-oriented terminology, such functions might be called *methods*. Requiring separate versions of these methods for each different type of object seems ungainly, and is out of step with informal mathematical convention.

As a result, Aczel and Barthes became interested in inheriting particular methods defined on one class of objects for other objects in the child classes of the original class. The approach taken involved setting up a forest of class definitions where an object in a child class could be coerced back into its parent class by forgetting some extraneous information. As the connection with subtyping became evident, it began to seem preferable to concentrate on the coercions themselves and to consider all types to be classes. This is the background against which my own work on coercions was developed.

4.1.2 Coercions in informal mathematics

Although mathematicians may not consciously be aware of coercions, if one approaches the language of informal mathematics from a type-theoretic perspective, one can make the case that their implicit use is one of the most powerful and useful features of this language.

If one applies the type-theoretic perspective, within which terms have particular types, to an average piece of informal mathematics, then at first sight the mathematics may simply seem to be ill-typed. Take, as an example, the number 5. In different circumstances it may be considered to be a natural number, an integer, a rational, a real, or even a complex number. However, if these types of number are specified in a formal framework, then the representations of 5 that have these types will be different objects. There seem to be at least five different representation of the number.

One can take the view that one is *overloading* the particular name "5"; that there are indeed many different objects, but the same name is being used to refer to each of them. Which particular object is being referenced in any given circumstance can be discovered by examining the context in which the name is used. Similarly, there are a lot of different objects with the names "4", "3", *etc.*

However, in itself, this view of overloading is unrevealing as it makes no connection between these different things named "5". I would argue all are supposed to be differently typed instances of the *same* underlying object. One way to persuade oneself of this is to observe the relationships that exist between these differently typed representations. The general relationships work not only for 5 but also for 4, 3, *etc.* Coercions encode these relationships in order to explain the meaning of the overloading.

One should find that any natural number, such as the one denoted by "5", can be made into an element of one of the other types mentioned by means of

a canonical construction. Of course, the exact details of the construction will depend on the implementations chosen for these types of numbers. To give an example, if an integer is to be represented as the difference $a - b$ between a pair of naturals (a, b) then one might turn the natural n into the integer $(n, 0)$. This is the process of *coercing* the natural number to be an integer. The canonical function that is applied to move between these types is called a *coercion*.

As another example, consider the construction of the set of binary mappings over some given set. There seems to be a constructor that takes a set as an argument and returns the set of binary mappings over that set as a result. However, if one has some group G, then it would be quite acceptable to a mathematician to talk about binary mappings over G, even though G is a group and not a set. Similarly one might talk about elements of G, even though a type-theorist might object that it is sets, not groups, that have elements. I claim one reason that these idioms do not seem unreasonable is that any group can be coerced to be a set by means of a canonical operation that extracts its underlying carrier. Inserting this coercion into the formal expressions would recover the well-typedness of the idioms.

4.1.3 Implicit syntax

One way to use coercions would be to incorporate them into the type theory itself. Thus the reductions permitted in a type theory could be defined relative to a context of declared coercions. In fact, since coercions implement a notion of abbreviation, much as identifier definition does, it can be persuasively argued[Luo97] that their use should be considered at the metalogical level of a logical framework within which type theories may then be defined, rather than at the object level within a particular type theory.

Alternatively, coercions can also be thought of as providing what is termed an

implicit syntax[Pol90] for expressions in an existing type theory. In such a syntax, some information (in this case, coercions) that may be reconstructed from the context can be omitted. In the case of an implicit syntax determined through coercions, the idea is that having defined a coercion $\kappa : S \to T$ one may use $x : S$ to represent the term $\kappa\ x : T$ in certain circumstances. For example, if f is a function which expects its argument to have type T, then $f\ x$ may be written as an abbreviation for the term $f\ (\kappa\ x)$.

From the viewpoint of the user, the implicit syntax thus acts as an alternative grammar for the type theory which *improves* the expressiveness of the theory (more terms have types), making it more succinct or intuitive, but without *increasing* this expressiveness (no additional types are inhabited) or introducing ambiguities (a unique term of the original type theory explains each piece of implicit syntax.) Such syntax can be seen as a step in bridging the gap between formalisations, which have the advantage of being essentially unambiguous and machine-checkable, and informal accounts, which have the advantage of being easy to read and understand.

Recall the mechanism known as argument synthesis that is already implemented in LEGO. This mechanism provides a concrete implementation of an implicit syntax that allows the omission of certain arguments in expressions; these arguments are later reconstructed automatically by the system (or *synthesised*) through unification.

The way in which I shall use coercions is through a corresponding mechanism of coercion synthesis. The user defines a collection of coercions which may then be left implicit in an extended syntax of expressions, in both input and output. On input, using information in the context of the expression, the implicit coercions will be automatically synthesised to recover a term of the original type theory. This term of the original type theory that exists inside the machine is said to

explain the shortened term of the implicit syntax. On output, the parts of terms that were synthesised are suppressed again.

I have made some changes to LEGO to incorporate coercion synthesis and hence implement an implicit syntax within the flavours of type theory used by LEGO. I call the resulting system "LEGO with coercion synthesis", or more often just "LEGOwcs". Other existing systems can be extended in very similar ways; Coq[Coq96] has also been extended with coercion synthesis by Amokrane Saïbi[Saï97a] to provide an implicit syntax for its underlying Calculus of Inductive Constructions[Wer94].

4.2 The system LEGOwcs

4.2.1 Overview

Recall that LEGO works using a context of identifier definitions and declarations. In LEGOwcs, some context entries may also be tagged as *coercion definitions* when they are first entered. The list of coercion definitions will represent a set of edges in a directed *coercive graph*. Paths through this graph constitute the *coercions* that may be *synthesised* within expressions and left implicit by the system in input and output. I shall use the terms "coercion" and "path" interchangeably.

The nodes between which a path runs are types; they are the domain and codomain of the type of the coercion function. I refer to these as the *source* and *target* types of the coercion. The source is the type a term needs to have for a coercion to be synthesised for it and applied to it, and the target is the type it will have after the application of the coercion. Each node of the graph is a specific type, and each path represents a specific function between its source and target types.

Limitations on the properties that the coercive graph is permitted to have force restrictions on the combinations of coercion definitions that will be accepted by the system. The main limitation is a requirement of coherency, explained in the next subsection.

Coercion definitions may be parameterised, and therefore one definition may introduce arbitrary numbers of nodes and edges into the graph. However, although infinite, the graph has a finite description; the list of coercion definitions gives rise to a finite collection of parameterised paths, built from the finite composition of the parameterised coercion definitions. This description may be used by deterministic algorithms to check that a graph has the required properties, and to synthesise the coercions left implicit in expressions.

Although the coercive graph may be thought of as being regenerated in its entirety every time the list of coercion definitions is changed, it is maintained incrementally in practice, since within LEGO, context entries (and hence coercion definitions) are introduced, and removed, one at a time.

There are some other general features of the LEGOwcs system which were decided on because of a variety of reasons to do with pragmatism, efficiency, and safety. These decisions and the alternative approaches that could have been taken are explored in chapter 5. The fundamental features of the approach used in LEGOwcs are as follows. Paths through the graph are constructed statically at the time that coercions are defined, not dynamically at the time of synthesis. Coercions are considered applicable only if the types involved match at a (mainly) syntactical level; the weaker requirement of convertibility is not sufficient. Finally, in all circumstances, if more than one coercion could be synthesised by the system, then the possibilities must be coherent in the sense defined in the next subsection. This allows a non-ambiguous form of "multiple inheritance."

4.2.2 Coercion definition and coherence

To enter a coercion definition into the context, the user needs to provide an identifier and a term. This term must be a function. It may take a number of initial arguments that can be reconstructed by LEGO's argument synthesis algorithm when the function is applied; these parameterise the coercion definition. The type of the first non-synthesised argument is the source type (or the parameterised class of source types) of the coercion definition. The type of the remaining function body is the target type (or the parameterised class of target types); this may depend on the source, and can be explicitly specified by the user through LEGO's type-casting mechanism if so desired. Thus the most general type for a coercion definition is

$$\Pi x_1 \mid p_1 . \Pi x_2 \mid p_2(x_1). \ \ldots \ \Pi x_n \mid p_n(x_1, \ldots, x_{n-1}).$$
$$\Pi x : S(x_1, \ldots, x_n). T(x_1, \ldots, x_n, x)$$

where $p_1 \ldots p_n$ are the (parameterised) types of the parameters, S is the (parameterised) source type, and T is the (parameterised) target type.

When a new coercion definition enters the context, new coercions (paths through the coercive graph) are generated from it. The simplest new paths are all of the single edges which result from the arbitrary instantiation of the parameters of the new definition. Also, existing paths to the source types of these new edges, and from their target types, are composed with these edges, to make new paths.

If competing paths are produced (ones that run between the same source and target types), then these paths are allowed only if they are extensionally equal on the source type. By this I mean that their applications to an arbitrary argument of this type should be convertible. I call the competing paths *coherent* in such a circumstance. A coercive graph is coherent if all competing coercions within it are. New paths which do not compromise the coherency of the coercive graph are allowed to be added. However, if a new path competes with an old one and they

are not coherent, then the coercion definition which resulted in the competing path being generated is disallowed. This generation of paths and checking for coherence is performed statically, at the moment that a new coercion definition is made.

In practice, one cannot generate (and check the coherence of) all the new paths individually, since the definitions may be parameterised over arbitrary types. Instead, the implementation works at the most general level of parameterisation of paths and definitions that it can. Since there are only a finite number of definitions, and each can be used only once in each path, only a finite number of combinations of these into parameterised paths is ever necessary. However, the process is most clearly understood through the preceding explanation at the level of the possibly infinite collection of individual non-parameterised paths. The details of the algorithms implemented will be given in section 4.4.

In order to keep terms unambiguous under all combinations of coercions, it is understood that every node of the coercive graph has an empty identity path defined upon it, and new coercions must be coherent with these identity paths. In this way we require that every cyclic path in the graph must be convertible with the identity. Although a cyclic path would never be synthesised directly, this prevents two successively synthesised halves of a cycle that act on some term from introducing any ambiguity.

4.2.3 Special flavours of coercion

Normally, coercions run between nodes that are particular source and target types. This is because generally when it comes to doing synthesis, LEGOwcs knows that it has a term with the source type where one with the target type is required. However, sometimes the precise target type that the term must have after it has been coerced may not be known. One may just know that one wishes

it to have one of a general class of types.

One such class is the class of kind types. Recall that in the terminology of LEGO a kind is any term which may appear on the right-hand side of a typing judgement; thus the types of kinds are Prop, Type_i and Type.

I thus introduce a special node into the coercive graph representing the class of all kind types. Coercions with this node as a target are of a special flavour called *kind-coercions*. They are the paths which end with an edge formed from a coercion definition that has been explicitly specified to be of this flavour. Whenever something that is not a kind is used as if it were one (for example, if it is found on the right-hand side of a typing judgement), LEGOwcs will search for a kind-coercion to apply in order to make the resulting expression well-formed.

It is these coercions which allow the treatment of composite objects, such as instances of groups or sets, as types over which one can quantify and range; if G is a group, we may also write G to represent its carrier type, projecting this by an implicit kind-coercion. Competing coercions from a source type to the special node of all kind types must be coherent, just as is the case for other targets.

The other special node in the graph represents the class of functional types or Π-types. Functions are terms which may be applied to other terms. Π-*coercions* are the flavour of paths with this node as a target, and they end with an edge formed from a coercion definition explicitly specified to have this flavour. Whenever something that is not a function is applied to an argument as if it were one, LEGOwcs will search for a Π-coercion to apply in order to recover the well-formedness of the expression.

This flavour of coercion is useful in formalising mathematics in type theory because proof information is often bundled up with functions (for example, to prove that a function is a set mapping, or an isomorphism), but often one still wishes to apply the composite object directly to arguments.

Π-coercions must also cohere when they compete. A less strong coherency requirement that allowed the existence of more then one Π-coercion from an individual type to differently typed functions would have been possible, but I did not attempt to implement this since I had never wanted a term to represent more than one function in my own work. (However, it may be that such an ability would be useful if one were to pursue the use of coercions to help with a certain form of overloading that is described in subsection 5.5.1.)

4.2.4 Syntactic matching of types

Whenever the system attempts to synthesise a coercion, it is always the case that we have an expression that requires some subterm to have a different type in order for the expression as a whole to be well-formed or well-typed. Therefore the proof-checker will need to decide whether or not the type of some term matches the source or target type of some coercion, and I must explain what I mean by matching.

Although any term in UTT may have many different types, the LEGO system provides each term with a unique principal type, which can be computed from the structure of the expression and from the principal types of its subterms, and changed by the user by means of an explicit type-casting if so desired. When talking about *the* type of some LEGO term as if it is uniquely defined, I will mean this principal type.

Two types are said to match for the purposes of coercion synthesis only if they have the same syntactic form, rather than the weaker condition of the types being convertible. The reasons for this are discussed in chapter 5.

The relation is defined at the level of the well-typed explicit syntax that underlies the system, but the user can consider the syntactic matching to take place at the level of the implicit syntax they read and write. The precise details

of the matching algorithm will be delayed until section 4.4; in brief, other implicit coercions and arguments may be synthesised in the types themselves before the comparison is made, and also the applications of these implicit coercions may be β-reduced, since these applications are invisible at the level of the implicit syntax.

Coercions, and hence the source and target types that can be matched, can be parameterised. Values are assigned to parameters as is required by the matching process; after a parameter has been assigned a value, then for another term to match this parameter, it must match the assigned value. Again, the precise details of the algorithm can be read in section 4.4.

4.2.5 Coercions in the context

In this section I explain how coercions work in relation to the LEGO context, and describe the commands used to work with them.

A coercion definition is entered into the context as a specially tagged entry. Before the definition is accepted, LEGOwcs checks that the term defined is a function, that all its parameters may be synthesised when it is applied to an argument, and that the graph that results from adding this new definition does not contain incoherent paths. If it fails any of these checks, then the definition is disallowed. Once accepted, the identifier defined can then be used in expressions in the normal way, independently of the generation of paths through the coercive graph that it also causes.

In the development, the definition of a coercion κ with definiens F of functional type $\alpha \to \beta$ is presented as

▷ Define coercion $\kappa = F : \alpha \to \beta$

If κ is to be a kind coercion, then also the target type (in this example, β) must be a kind for the definition to be accepted. Such a definition is presented as

▷ Define kind-coercion $\kappa = F : \alpha \to \beta$

Similarly, if κ is to be a Π-coercion then there is an extra requirement that it returns a function. If accepted, such a definition is written

▷ Define π-coercion $\kappa = F : \alpha \to \beta$

Coercion definitions (and all following entries) can be forgotten from the context, just as any other context entry may be. When this occurs, all of those paths generated since the coercion definition being forgotten are removed from the coercive graph.

I also need to explain how coercion definitions work in relation to the discharge of local identifiers. Coercion definitions may be made local. When they are discharged, then all paths that were generated from them are removed from the coercive graph, and all synthesised instances of these paths that are in the context become local definitions within the bodies of the expressions in which they occur.

Coercions may also rely on local declarations; when such declarations are discharged then they are thereafter treated as extra implicit parameters to the coercion, as one might expect. This is achieved by abstracting the coercion definition over the declarations in the standard way as for all discharges, except that the bindings are forced to be implicit even if the original declaration used an explicit binding.

When a discharge command affects the set of coercion definitions, then the graph is regenerated and coherency rechecked from the point of the earliest of the coercion definition that is affected. This is achieved by forgetting the paths that used the original coercion definitions from the coercive graph and then introducing the altered definitions sequentially, generating the paths in the standard way. Since the new definitions are more general than the older ones, a larger number of paths may be generated. If a problem concerning coherency results, then the discharge command that caused the problem is not allowed to take place.

There are also some new commands that enable the user to review what coercion definitions have been made and which paths have been generated. These are not used within the development; they are intended for use in an interactive session. I shall therefore describe them only briefly. The command "`Coercions`" lists all coercion definitions in the context. The command "`Paths`" lists all generated coercions. There are three more specialised versions of the "`Paths`" command to view particular coercions in the graph. "`PathsFrom` S" lists all coercions that could be synthesised from the particular type S, "`PathsTo` T" finds all those to the type T, and "`PathsBetween` S,T" returns all the coercions between S and T.

4.2.6 Coercion synthesis

I shall now describe all the circumstances under which LEGOwcs will attempt to synthesise a coercion. If any of the following expressions involving the subterm X with principal type S fail to be well-formed or well-typed within the standard grammar, then the system attempts to synthesise a coercion $\kappa : S \to T$, and then replaces X with $\kappa\ X$ in order to produce a well-typed expression.

The first sort of expression is very common and is the canonical case in which a coercion is synthesised. These are the sort of syntheses that will allow one to talk about binary mappings over a group rather than a set. A function is applied to an argument that has the wrong type, and a coercion is used to correct the mismatch.

The second sort of expression allows the invocation of a coercion through an explicit type-casting. The last two are also connected with type-casting; recall that a pair has a non-dependent Σ-type unless is explicitly given a dependent one. Although this looks like a type-casting, it is handled slightly differently within LEGO, and so must be dealt with separately in LEGOwcs.

- An application, $f\ X$ where the principal type of the argument to f is T

- A type-casting, $(X : T)$
- A casting within a dependent Σ-type, $(X, b : \langle \Sigma x : T \rangle\ t)$
- A casting within a dependent Σ-type, $(a, X : \langle \Sigma x : s \rangle\ T[x/a])$

Certain declared flavours of coercions can also be synthesised in additional circumstances, as well as those outlined above. A Π-coercion from S is synthesised in the circumstance when X is not a function, but is being applied to another term as if it were.

- An application, $X\ a$

Kind-coercions from S are synthesised in many circumstances when X is not a kind, but a kind is nonetheless expected in order to provide a well-formed expression:

- An abstraction, $[\lambda x : X]\ t$, $\{\Pi x : X\}\ t$, $\{\Pi x : s\}\ X$, $\langle \Sigma x : X \rangle\ t$ or $\langle \Sigma x : s \rangle\ X$
- A type-casting, $(x : X)$
- A casting within a dependent Σ-type, $(a, b : X)$

Within LEGOwcs coercions are also allowed to be synthesised in other circumstances where interactive proof commands expect an expression to have a certain type or one of a certain class of types. Since my presentation does not make the details of any interactive proofs explicit, these syntheses need not be considered by the reader of the development. However, for the sake of the interest of an existing LEGO-user, they are as follows.

- A refinement "`Refine` X", where the current goal is T.
- A goal command, "`Goal` X"; a subgoal command, "`Claim` X"; and a command to use an equivalent goal, "`Equiv` X", can all synthesise kind-coercions.

- Also, some of the new commands specific to listing coercions in LEGOwcs, "`PathsFrom` X", "`PathsTo` X" and "`PathsBetween` X,X", can also synthesise kind-coercions.

4.3 Examples

This section demonstrates how coercions may be used in practice with some simple examples that take as little time as possible to set up. A broader range of useful and interesting uses can be shown once one has some framework of mathematical concepts implemented that one can work with respect to. Some illustrative examples from the main Galois development include the interaction between subsets, sets, and their elements, fields considered as vectorspaces over themselves, and natural numbers used as canonical indexing sets. These can be found in chapters 8 and 9.

I start with some abstract examples in order to illustrate the basic mechanisms of coercion definition and synthesis.

4.3.1 Abstract examples

The first example is a simple coercion whose synthesis is caused by a function application. This canonical case is probably the most common one in practice.

▷ Introduce $f : C \to D$ and $b : B$

▷ Define coercion $\kappa_1 = \ldots : B \to C$

Now κ_1 allows objects of type B to be used where an argument of type C is expected. The coercion κ_1 is synthesised but is kept implicit in the output.

▷ $f\ b : D$

▷ $f\ b \simeq f\ (\kappa_1\ b)$

Now I add a second coercion definition:

▷ Define coercion $\kappa_2 = \ldots : A \to B$

I can force a coercion to be synthesised with an explicit type-casting:

▷ Introduce $a : A$

▷ $a : B$

Also, the existing coercion definitions will be composed to generate a third coercion $\kappa_1 \circ \kappa_2$ from A to C.

▷ $f\ a : D$

▷ $f\ a \simeq f\ (\kappa_1\ (\kappa_2\ a))$

If I define a kind-coercion from B, I can consider any object of type B as a type in its own right.

▷ Define kind-coercion $\kappa_3 = \ldots : B \to \mathsf{Type}_0$

▷ Introduce $x : b$

Here, using b as a type has caused the coercion κ_3 to be implicitly synthesised:

▷ $x : \kappa_3\ b$

Other uses can also cause the same coercion to be synthesised to turn b into a type. Note that also a will be a type through the coercion $\kappa_3 \circ \kappa_2$.

▷ $a \to b \# b : \mathsf{Type}_0$

Finally, I shall define a Π-coercion to allow objects of type A to be used as functions.

▷ Define π-coercion $\kappa_4 = \ldots : A \to C \to A$

▷ Introduce $c : C$

▷ $a\ c : A$

I end these abstract examples with a deliberately complicated case that shows all four coercions being implicitly synthesised in the same very short expression.

▷ Introduce $X : a\ b$

▷ $X : \kappa_3\ (\kappa_2\ (\kappa_4\ a\ (\kappa_1\ b)))$

4.3.2 Concrete examples

I now outline a simple framework for mathematics to enable me to present some examples of coercion use in the formalisation of some algebra. Suppose that set, an additive abelian_group and field are all types, with coercions between them.

▷ Define coercion $\kappa_5 = \ldots$: abelian_group $\rightarrow$ set

▷ Define coercion $\kappa_6 = \ldots$: field $\rightarrow$ abelian_group

Defining a kind-coercion allows a set to be written to represent its underlying element type.

▷ Define kind-coercion $\kappa_{el} = \ldots$: set $\rightarrow$ Type_0

Suppose now that for S : set, we have defined the set of binary operators on S.

▷ Define $\mathsf{op}_2 = \ldots$: set $\rightarrow$ set

Every G : abelian_group has an associated addition mapping, represented by the symbol $+|G$. The group argument G to $+$ is implicit since in normal use we will suppress it and allow it to be synthesised from the arguments to which $+$ is applied. The coercion κ_6 allows a field to "inherit" this addition mapping.

▷ Introduce F : field

▷ $+|F$: $\mathsf{op}_2\ F$

If you look closely at the above typed expression, you will see that all three coercions introduced so far need to be synthesised within it in order to make it type-check:

▷ $+|F \simeq +|(\kappa_6\ F)$

▷ $\mathsf{op}_2\ F \simeq \kappa_{el}\ (\mathsf{op}_2\ (\kappa_5\ (\kappa_6\ F)))$

We would expect to be able to apply operators such as $+$ to arguments directly. In fact, we would expect to write it infix between two arguments, and this is allowed in the development using a pretty-printing directive. But leaving matters of its fixity unaddressed for now, we will also need a Π-coercion in order to extract

the applicable function from the object +, which does not have a function type.

▷ Define π-coercion $\kappa_{ap} = \ldots : \{\Pi S \,|\, \mathsf{set}\}\ (\mathsf{op}_2\ S) \to S \to S \to S$

Here κ_{ap} is our first example of a parameterised coercion, parameterised over the argument $S : \mathsf{set}$. Its source type is $\mathsf{op}_2\ S$ and its target type is a functional one, $S \to S \to S$. Again coercions have been used to write these expressions: the source type is actually $\kappa_{el}\ (\mathsf{op}_2\ S)$ and the target type actually $(\kappa_{el}\ S) \to (\kappa_{el}\ S) \to \kappa_{el}\ S$. We can now apply + as expected:

▷ Introduce $x, y : F$

▷ $x + y : F$

Again it may be revealing for the sake of this example to examine what has been synthesised in the above expression.

▷ $x + y \simeq \kappa_{ap}\ (+|(\kappa_6\ F))\ x\ y$

▷ $F \simeq \kappa_{el}\ (\kappa_5\ (\kappa_6\ F))$

In this case both implicit arguments and implicit coercions have been synthesised to expand the natural abbreviated expression to a longer and more formal form.

To continue this example, we consider vectorspaces defined over a field. A vectorspace is an abelian group together with a scaling operator $*$ that obeys certain axioms, so another coercion is used to formalise that relationship.

▷ F-space : Type_1

▷ Define coercion $\kappa_7 = \ldots : F\text{-space} \to \mathsf{abelian\_group}$

▷ Introduce $V : F\text{-space}$ and $v, w : V$

▷ $*|V : F \to V \to V$

(In fact, in the development, $*$ will be a mapping rather than a plain function, but I did not wish to complicate this example by using a more general Π-coercion than κ_{ap} that would allow such a mapping to be applied.)

Once more, note that kind-coercions are being synthesised for F and for V in the above expressions. A reader does not tend to notice these coercion syntheses

unless they are made explicit, since they are not essential for an understanding of what is presented. This, I feel, is how it should be, and this scenario will be common in the Galois development. The reader sees expressions that are natural in appearance and familiar from informal style, but whenever necessary these can be translated back to the more cumbersome fully formal and machine-checkable expressions.

The machine does such a translation for all statements in order to get them into a language that it understands. A human reader rarely needs to consider the details of this translation. This is because the objects behave in a way that is in accord with the reader's mathematical intuition. There is an "obvious" translation back to formal type theory via coercions; each object in the expression may be turned into something of the correct type using canonical methods. Since the coherency of the coercive graph guarantees the expressions are unambiguous, the reader knows that the translation the machine has used to check the mathematics *is* the obvious one.

I return from that tangent to the example. Since + can synthesise different abelian groups from the context in which it is used, we have now enabled some form of safe overloading.

▷ $(x + y) * (v + w) : V$

▷ $(x + y) * (v + w) \simeq (\kappa_{ap}\ (+|(\kappa_6\ F))\ x\ y)\ (*|V)\ (\kappa_{ap}\ (+|(\kappa_7\ V))\ v\ w)$

After the field F is discharged from the context, the coercion κ_7 will be parameterised over this argument.

▷ Discharge w, v, V, y, x, F

▷ $\kappa_7 : \{\Pi F \,|\, \mathsf{field}\}\ F\text{-}\mathsf{space} \to \mathsf{abelian\_group}$

To conclude, I consider two final ideas: dependent coercions, and coherency checking. Any field may be considered to form a vectorspace over itself; the scaling operator for this vectorspace is simply multiplication in the field. We can

formalise this use with a final coercion:

▷ Define coercion $\kappa_8 = \ldots : \{\Pi F : \mathsf{field}\}\ F\text{-}\mathsf{space}$

This is a dependent coercion; the target type of the coercion definition depends on the value of the source argument. The idiom will allow us to use scaling on elements of a field:

▷ $x * y : F$

▷ $x * y \simeq *|(\kappa_8\ F)\ x\ y$

Before κ_8 was accepted as a coercion, however, LEGOwcs had to do some coherency checks. This is because there are now two distinct paths in the group between fields and abelian groups (and hence also further objects such as sets and types.) The one already explicitly declared is κ_6, but a new one has now been generated that goes via vectorspaces, $\kappa_7 \circ \kappa_8$. In order for these coercions to be coherent, we require that for arbitrary fields F, the following conversion judgement must hold:

▷ $\kappa_6\ F \simeq \kappa_7\ (\kappa_8\ F)$

For sensible definitions, this ought to be the case, since the additive abelian group underlying the field and the one underlying it when it is considered as a vectorspace over itself are one and the same. LEGOwcs is able to establish this by expanding the definitions of the coercions and other terms. (I have suppressed those definitions for the purposes of this example, but their details in the Galois development can be found in Appendix section B.7.2.1.)

4.4 The LEGOwcs implementation

The above description of the system and examples of its use should suffice to understand the concepts involved and to read a formalisation that makes use of coercions. This section examines the details of the algorithms used within the LEGOwcs implementation. To completely understand the behaviour of the

system, one needs to understand the basic algorithm for matching types, and the further algorithms that make use of this first one in order to synthesise the coercions left implicit in an expression, and to add newly generated coercions to an existing graph following a new coercion definition.

The coercive graph is represented internally as a list of parameterised paths. Each parameterised path represents all of the specific coercions that may be obtained by instantiating its parameters. Each is generated by the composition of one or more coercion definitions at the most general level of parameterisation under which they will compose.

Dependent coercions, in which the target type of a coercion may depend on its source value, may appear to complicate matters. In fact, they do not cause any extra implementation difficulties. In all such cases, the argument is matched as if it were another parameter to be assigned. In the case when the argument is known, as in the coercion synthesis process, the match is trivial; the argument parameter is immediately assigned the known value. However, treating the argument as a general parameter also allows the matching needed for comparing and composing coercions that is described in the later sections to transparently generalise to dependent coercions. Thus in the following, when I write of a coercion between the parameterised types S and T, then, if necessary, I include the argument of type S as an extra parameter to T to cover the dependent case.

4.4.1 Matching of types

This algorithm takes a number of pairs of terms, which may be parameterised, and attempts to find a common parameter assignment under which the first term in each pair is matched with the second. It works by recursion over the structure of the terms it has to match, matching each pair in turn and building up the

parameter assignment as it goes.

Let the parameters be $x_1, x_2, \ldots, x_n$ with types $p_1, p_2(x_1), \ldots, p_n(x_1, \ldots, x_{n-1})$. Let the parameter assignment calculated be represented as a partial function σ, so $\sigma(x_i)$ is the assignment derived for the parameter x_i, or is undefined if no such assignment has yet been derived. The completely undefined assignment is written $\emptyset$, and the assignment that assigns the term X to the parameter x_i will be written $x_i \mapsto X$. If no valid assignment can be found the matching function will indicate its failure by returning an invalid assignment represented by $\perp$. Non-conflicting assignments can be combined, so $\sigma_1 + \sigma_2$ is a valid assignment so long as for all i, if $\sigma_1(x_i)$ and $\sigma_2(x_i)$ are both defined, then they are equal. If the assignments conflict, their combination is $\perp$.

The value of a parameterised term t under the parameter assignment σ is written $\sigma[t]$. This is just t with each instance of x_i replaced by $\sigma[\sigma(x_i)]$ if the assignment is defined, and left unchanged if it is not. The way in which assignments will be built up ensures that the implied recursion always bottoms out, since the combination $\sigma + (x_i \mapsto X)$ will be disallowed when $\sigma[X]$ contains x_i.

If the terms s and t match and the calculated assignment is σ, then this will be written $s \equiv t = \sigma$. In such a circumstance then at the level of the implicit syntax, $\sigma[s]$ will be identical to $\sigma[t]$. If the terms don't match under any assignment then we will have $s \equiv t = \perp$.

The matching relation is then calculated by recursion:

- First of all, the matching relation is to be reflexive. Thus $s \equiv s = \emptyset$.
- The most complex case is matching against a parameter.
  - $x_i \equiv x_i = \emptyset$ already by reflexivity.
  - Otherwise, if $t \neq x_i$ but $\sigma[t]$ mentions x_i, then the match fails, and $x_i \equiv t = \perp$. (This rule ensures that no parameter will be defined in

terms of itself.)

- Otherwise, if $\sigma(x_i) = s$, then $x_i \equiv t$ reduces to $s \equiv t$.
- Finally, if $\sigma(x_i)$ is not yet assigned, then $x_i \equiv t = (p_i \equiv T) + (i \mapsto t)$, where p_i is the given type of the parameter x_i, and T is the principal type of t.

- Now I consider applications.
  - β-reduction of coercion applications is allowed. Thus $[\lambda x : T]\ e\ s \equiv t$ reduces to $e[s/x] \equiv t$ if the applied function was synthesised as a coercion.
  - Apart from this reduction, $f\ s \equiv g\ t$ reduces to $(f \equiv g) + (s \equiv t)$.

- All other matching cases occur when the same type-theoretic operator is being applied to matching constituent parts.
  - $s.\mathbf{1} \equiv t.\mathbf{1}$ reduces to $s \equiv t$.
  - $s.\mathbf{2} \equiv t.\mathbf{2}$ also reduces to $s \equiv t$.
  - $(s_1, s_2) \equiv (t_1, t_2)$ reduces to $(s_1 \equiv t_1) + (s_2 \equiv t_2)$.

  In any of a pair of λ, Π or $\Sigma-$bindings, if the first has domain s_1 with co-domain s_2, and the second has domain t_1 with co-domain t_2, then the match reduces to $(s_1 \equiv t_1) + (s_2 \equiv t_2)$ after an appropriate renaming of bound variables. (In fact, LEGO uses de Bruijn notation internally and keeps track of names only for output purposes.) Thus the sort of binding used (implicit or explicit) is ignored. Thus, for example,
  - $[\lambda x : s_1]\ s_2 \equiv [\lambda y \,|\, t_1]\ t_2$ reduces to $(s_1 \equiv t_1) + (s_2 \equiv t_2[x/y])$.
  - $\{\Pi x \,|\, s_1\}\ s_2 \equiv \{\forall y : t_1\}\ t_2$ reduces to $(s_1 \equiv t_1) + (s_2 \equiv t_2[x/y])$.
  - $\langle \exists x : s_1 \rangle\ s_2 \equiv \langle \Sigma y \,|\, t_1 \rangle\ t_2$ reduces to $(s_1 \equiv t_1) + (s_2 \equiv t_2[x/y])$.

4.4.2 Coercion synthesis

Coercion synthesis is attempted in a variety of occasions within LEGOwcs. The details of these occasions were explained in section 4.2.6. In each case, before the introduction of coercions into LEGO, the circumstance would have resulted in an error indicating a problem with an ill-typed or an ill-formed term. In LEGOwcs, the proof-checker tries to synthesise a coercion to recover from the problem.

There are three flavours of coercion synthesis that are performed.

- Standard coercion synthesis to coerce X of type S into $\kappa(X)$ of type T
- Kind-coercion synthesis to coerce X of type S into a kind $\kappa(X)$
- Π-coercion synthesis to coerce X of type S into a function $\kappa(X)$

In all three cases we require a coercion κ whose parameterised source type will match S in the manner described above under some assignment σ. In the first case, the parameterised target type of κ must match T under the same assignment. In the second and third cases, there is no such extra matching requirement, but κ must be flagged as a kind-coercion or Π-coercion, respectively.

The synthesis proceeds by trying each coercion in the graph in turn to see if it will meet the relevant requirements. If no such coercion can be found, then the synthesis is unsuccessful and the original error corresponding to the circumstance that prompted LEGOwcs to look for a coercion occurs. If instead a coercion is found that meets the requirements it is returned immediately, since the manner in which the coercive graph is constructed ensures there is only one possible synthesis (at least up to type-convertibility.) The checking needed to ensure this is described in the next section.

4.4.3 Adding new paths to the coercive graph

A new coercion definition may give rise to many new parameterised coercions (paths through the coercive graph.) The way in which these are generated is described later in this section. First I consider what happens to each new individually generated coercion.

The first requirement on an acceptable path is that it may not contain more than one use of any parameterised edge of the graph. That is to say, it may not be formed from a composition involving two edges that both result from possibly different parameterisations of the same coercion definition.

The reason for this is that some definitions are sufficiently generic that they could be composed with themselves or with each other arbitrarily many times under different parameterisations. For example, consider a coercion definition $\kappa : \{\Pi t \,|\, \mathsf{Type}_0\}\ (t \# t) \to t$. Given any $T : \mathsf{Type}_0$, a path from $(T \# T) \# T \# T$ to T can be formed by composing two instances of κ under the parameterisations $t \mapsto T \# T$ and $t \mapsto T$. Clearly this sort of composition could be extended on the left indefinitely.

My solution, to not consider any path that involves two instances of the same definition, is a straightforward one, which makes it easy to work with in practice. If a user ever does wish to allow a path involving two copies of the same coercion definition, this is easily accomplished by entering a second copy of the same coercion definition into the context.

The next thing to consider is whether a coercion definition κ can form a cycle under some parameterisation. If its source and target can be matched, $S \equiv T = \sigma$, then $\sigma[\kappa]$ is regarded as competing with the identity function on T, and coherency checks and decisions about what action to take follow the pattern for competing paths that is described below, given that one considers there to be an empty identity path defined on every node of the coercive graph.

Apart from these empty cycles, a coercion may also be found to compete with other paths that are already in the coercive graph. For every existing parameterised path κ' from S' to T', if there is a match $S \equiv T = S' \equiv T' = \sigma$ then we say that κ competes with κ' under the assignment σ. For these paths to be coherent, it is required that $\sigma[\kappa]$ is convertible with $\sigma[\kappa']$.

If the new path is a kind-coercion (or respectively, a Π-coercion) rather than a normal coercion, then it must still pass the first coherency test. In addition, coherency is also checked against all kind-coercions (or respectively, Π-coercions) from a source S' that can match the source S of the new path. The idea is that both the coercions run to the node representing all kinds (or respectively, all Π-types) and so the targets of these coercions already match. If this additional test is successful, the new coercion does not introduce incoherencies.

If a competing path is found that is incoherent with the new path, then the coercion definition that generated the new path is disallowed and no paths generated from it are allowed to be added into the graph. Thus all paths generated from a new coercion definition must be coherent with existing paths for that definition to be acceptable.

If all paths that compete with κ are coherent with it, then κ *can* be safely introduced into the graph. However, if under all parameterisations κ already has a competitor, then it will never be synthesised in the current implementation. I make some effort to determine whether or not this is the case, to avoid introducing a useless path into the graph. In fact I only consider the case when the same existing path competes with the new one under all assignments, since this is relatively easy to check. This approach of not introducing redundant coercions helps to keep the graph size small, but it does have one disadvantage which will be explored within section 5.2.6.4.)

4.4.4 Generating new paths from a coercion definition

The first parameterised path generated by a new coercion definition is that formed by the coercion definition itself. This may be seen as the path consisting of a single edge.

The other new paths generated are those created by joining existing paths to the source and from the target of this path to either or both of its ends. Since a coercion definition may feature only once in any path, this may be achieved by first attempting to compose all existing paths on one side of the coercion definition, and then attempting to compose all previously existing paths on the other side of all the new paths so far generated.

To compose two parameterised coercions, the target of the first has first to be matched with the source of the second. If there is no assignment under which a match can be made, the paths cannot be composed.

Having produced an assignment σ that allows a match, it must then be checked that all the parameters (including the the source argument) of the second coercion can be expressed in terms of those of the first coercion (again, the source argument is included as a parameter.) This can be done by examining the values of these parameters under the assignment σ. If some parameters cannot be expressed in this way, then the composition will not be carried out, because the resulting coercion would have some unsynthesisable parameters. Otherwise, the composition is parameterised by the remaining parameters unassigned by σ, and its body is equal to the functional composition of the assignment σ applied to each of the two component paths.

Chapter 5

Further coercions

This chapter takes a broader look at coercions, exploring some other ways in which the concept can be used. I explain the design decisions taken in the implementation of LEGOwcs and the reasons for choosing them over their alternatives. I also consider the status of coercions relative to related concepts in type theory such as subtyping and overloading.

5.1 Putting coercions into type theories

The LEGOwcs system uses coercions to provide a new implicit syntax for expressions. However, the type theory underlying this syntax, and the reduction and conversion rules that govern it, remain unchanged. If one considers the system at the level of the implicit syntax, as a user is inclined to do, then one may formulate new formal rules and judgements at this level; but there is a simpler formal system underneath without coercions that gives rise to these.

Since developments in LEGOwcs have a simple translation back into the trusted type theory of the traditional LEGO system, one may use the extended system and be confident about the results it proves without having to perform a full-scale investigation of its meta-theoretic properties - these can be inferred

from those of the simpler system. However, the terms involving synthesised coercions with which the proof-checker has to deal can be very large, although this may not be evident to the user, who reads only the more concise implicit syntax. Working at two levels in this way and translating between them can introduce unnecessary inefficiencies in the proof-checker itself.

An alternative approach is to incorporate coercions into the type theory at the level at which it is formulated. This means the reductions and judgements of the type theory are then defined relative to a collection of declared coercions. One could extend the rules for any given system with a particular formulation of coercions, and investigate the effects this might have. Gilles Barthes, amongst others, has investigated such an approach[Bar96]. The resulting type theory could then be implemented directly from this formal basis. (I am not aware of any such implementations.)

However, the mechanism of abbreviation allowed by coercions seems to be most naturally considered at the metalogical level, much like the abbreviating power of definitions. Zhaohui Luo has investigated the use of coercions at the level of a logical framework[Luo97]. Within a general logical framework the more specialised theories such as some of the common extensions to Martin-Löf Type Theory[MLof84] or the Calculus of Constructions[CH88] can be formed by defining appropriate type constructors. By making these definitions within a logical framework extended with a system of coercions, one can introduce coercions into any of these theories, and reason about the effect this will have on its proof properties using the logical framework.

The power of such a metalogical approach can be illustrated with the observation that all of the many forms of coercion synthesis found in LEGOwcs can be explained and justified at the level of a logical framework in terms of the one most simple and fundamental use of coercions; changing the type of a function

argument in order to match that required by the function.

As an example, consider an informal presentation of a logical framework (such as LF[Luo94]) with a single sort denoted $*$, product types written as $(x : S)\,T$ or as $(S)\,T$ if there is no dependency, and abstractions with such types as $[x : S]\,y$. One can add a reduction rule to the framework so that applications may utilise coercions: if f has type $(x : S)\,T$, and a has type S', and there is a coercion κ from S' to S, then the application $f\ a$ is computationally equal to $f\ (\kappa\ a)$.

In such a framework, this single coercive reduction rule can account for the other flavours of coercion that needed to be considered separately when working with the direct syntax in LEGOwcs. For example, the Π-type constructor will have type $(S : *)\,((S)*)\,*$. The kind-coercions that allowed Π-abstractions over non-types just become regular coercions into $*$; Π-coercions are regular coercions into the parameterised class of types defined by the Π-type constructor; type-casting a term to have type S can be represented as the application of the identity function on S, and so on.

Coercions can also be used as a framework for universe subsumption by considering a ladder of coercions ι_n for natural numbers n, each from the nth to the n+1st level in the universe hierarchy. Luo discusses these ideas further in [Luo97].

5.2 Design decisions in LEGOwcs

In chapter 4 I described the make up of the LEGOwcs system and gave details of how it was implemented. Many of the decisions that were taken in formulating the system are interrelated in a complicated way, and were influenced by both theoretical and pragmatic motivations. Some of the decisions taken may on first sight not appear to have been the most natural choices.

In this section I look at these decisions, exploring the reasoning behind them

and looking at the alternatives that could be (or have been) used in principle (or in practice.) To begin, I review the priorities and motivations that were guiding me.

The implementation of coercions in the Coq proof-checker[Coq96] by Amokrane Saïbi[Saï97a] shared many of the same desires for practical checking and expressivity, and I note he made many similar choices. There are differences too, and I shall compare and contrast the approaches taken by the two implementations where this is the case.

5.2.1 Motivations

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 system therefore had to run relatively efficiently and not slow the operation of the existing machinery inordinately. Especially when dealing with large terms, the existing generation of proof-checkers are fairly demanding in terms of compute time and resources. Modifications which caused too much of a further slowdown would not be feasible to use for a large-scale development. This influenced the sort of matching I used and had a bearing on the decision to generate the coercive graph statically.

The implicit syntax that results from the implemented features had to be as flexible as possible and have the ability to express as many useful notations and to

synthesise as many convenient coercions as possible. This influenced my decision to allow coercions to be parameterised, and for paths between nodes to compete if necessary.

If a final formal development is to be usefully read, then it is important that the reader can parse the formal expressions and understand what they mean. This is the reason for using coercions in the first place - to provide an implicit syntax that eliminates some unnecessary clutter in order to allow some informal idioms. However, it is important that succinctness of expressions is not achieved at the cost of introducing ambiguity. Potential ambiguities are dangerous, for even if the writer of a development is able to avoid mistakes in their own understanding, a reader who is not as familiar with the material may be misled if the syntax reads in an ambiguous manner. This influenced the enforcing of coherency between competing coercions.

Finally, since my changes involve tinkering with the proof-checker itself and hence risk introducing bugs, it is important that a proof produced by the extended system should be easy to translate back into a form that can be checked by the original version or by other trusted proof-checkers; it must be believable[Pol97]. Since the system only implements an implicit syntax for an existing type theory, the translation is not difficult; one just makes all the coercions synthesised by the extended system explicit. This produces terms of the original type theory. As a proof of concept check, I performed this translation by hand using search and replace in the compiled files that LEGO produces for a small part of the development, and encountered no difficulties.

However, I note that making these coercions explicit may clutter and obscure the expressions that need to be read to understand what has been proved, because these expressions were originally formulated in the cleaner, kinder, implicit syntax. It might well be worthwhile, therefore, to include in any trusted system

the ability to suppress upon output those functions understood to be coercions. Thus while coercions need play no special role in formal checking, the fact that they do in informal understanding may mean that they are too important to ignore when a reader is trying to believe a proof.[Pol97]

5.2.2 Parameterisation

Parameterisation allows the simultaneous introduction of large classes of coercions into the graph. Although this does not affect the underlying rationale of the system, it has consequences for any implementation on a machine, as it produces a potentially infinite number of individual coercions.

Without parameterisation, a finite number of coercion definitions results in a finite number of paths, each of which can be considered individually and in sequence when coherency is checked, or when deciding whether something can be synthesised. With parameterisation, one can no longer act in so simple a manner. An algorithmic process is necessary to decide which instantiations of parameters will yield the coercions that are relevant to the given situation, and the need to make the process feasible will influence the other decisions taken. Therefore, it is important to justify the decision to make use of parameterisation.

The motivation for making the decision to allow parameterisation was my desire for flexibility and expressiveness. The importance of such things is clearly difficult to quantify. I have found parameterised coercions to very greatly increase the usability of the system. Examples will be found throughout the case-study later in this thesis. In fact, within the development I have undertaken, more than half of all coercion definitions are parameterised. For now, I will attempt a more general explanation.

The coercions that are left implicit to improve the readability of an expression act on objects of certain types. Parameterisation of coercions is necessary only if

these types are drawn from a parameterised class. For example, coercions from sets or groups do not need to be parameterised since these types are unparameterised. However, many interesting objects in algebra do have such parameterised types. One example is the type of vectorspaces parameterised over arbitrary fields (page 189.) A second is subgroups of arbitrary groups (page 186.) A third would be mappings over arbitrary sets (page 166); almost all Π-coercions tend to be parameterised. Lastly, some coercions act on elements of models of particular algebras (*e.g.* elements of some given set); these thus need to be parameterised over instances of the models in question (page 170.)

Parameterisation also offers a simple method for dealing with dependent coercions, where the target type depends on the source argument. An example of this would be to coerce a field into a vectorspace over itself (page 189.) Such dependencies can be dealt with by treating the source argument as an additional parameter in the target type. Thus an implementation of parameterisation is sufficient to handle such dependent coercions.

Since my system was built with the particular piece of mathematics that I am formalising in mind, I should make the disclaimer that what I have found in practice may not concord with other experiences. For example, it may be that algebra is unusual in its demand for parameterised coercions. However, I would guess that parameterisation is similarly useful in other domains. It also features in another practical implementation of coercions, that of Saïbi in Coq.

5.2.3 Graph generation

LEGOwcs maintains a representation of all paths through the coercive graph by generating new paths whenever the set of coercion definitions is changed. An alternative to static generation would be to perform some path-searching in the graph dynamically when a coercion needs to be synthesised.

A dynamic approach requires a different approach to coherency (see sections 5.2.5 and 5.2.6.) However, this is not necessarily a bad thing in itself. In some ways, a dynamic search allows more expressiveness and flexibility since there is more information to hand when synthesising a specific path than when generating a general parameterised one. It also allows the easy extension of existing coercions on base types to ones on inductive types and Π-types over these bases. This is not performed by LEGOwcs because I found it complicated static coherency-checking and have not found a need for it in my own developments.

Static generation is used in LEGOwcs because the amount of computation involved in the dynamic graph-searching that would be required to trace paths and synthesise coercions on-the-fly is very large. Each expression of average size contains several implicit coercions. In the interests of efficiency, it is preferable to do as much of the computation as possible once only, at the moment the coercion definitions are made, rather than repeating similar work every time an expression is type-checked. This is the same sort of reasoning as that which implies that compilation is usually preferable to interpretation when trying to implement an efficient programming language.

Faster machines and better algorithms might change things, but for the moment any pragmatic implementation would seem to need to do most of the work involving coercions statically at "compile-time" rather than dynamically at "run-time."

5.2.4 Matching

Coercions run between nodes in the coercive graph. These nodes represent particular types. Thus when synthesising a coercion for a subterm one needs to have some way of matching types to decide which coercions might be applicable. In

LEGOwcs, types must have the same syntactic form to match. A similar requirement of syntactic matching is also enforced in the Coq implementation. However, other sameness relations on types exist which may seem more natural to use. An obvious possibility is the more generous one of convertibility; the theoretical presentations of coercions by Barthes and Luo both use the convertibility relation. I should therefore explain why a different approach is used in LEGOwcs. There are four main reasons.

Firstly, there is an immediate efficiency concern. A comparison of syntax is quicker to perform than a check for convertibility. Although algorithmic tricks can be used to speed up convertibility checks, in the worst case one is required to reduce the types being compared to their normal forms. Since two matches (source and target) are necessary for each coercion in the graph, a single coercion synthesis involves a large number of matches. In turn, as previously mentioned, an average expression may require a number of such syntheses.

Secondly, although a statement of convertibility

$$s \simeq t$$

is decidable in UTT for simple types, it is no longer decidable when one or both types involve unknown term-variables

$$\exists x_1, \ldots, x_m.\ s(x_1, \ldots, x_m) \simeq t$$

since in general this requires higher-order unification, which is undecidable[Gol81]. When parameterisation is used, this sort of unification is necessary, since parameters might be assigned to arbitrary terms. Most matches will require only first-order unification, and there are tactics that can be used to solve the higher-order unification problem in certain cases[Nip90], but it can be seen that there are some fundamental problems here, and that even an incomplete implementation will have to do some complex unification computations.

Since I have decided that parameterisation is essential in a flexible and expressive system of coercions, and that a practical implementation must be reasonably efficient, convertibility seems to be unsuitable. Syntax-based matching does not present these problems, as parameter assignments can be calculated without difficulty for such a stricter matching of types, as described in section 4.4.1.

Thirdly, although a generous matching relation means it will be possible to synthesise coercions without further work in more cases, there is a related drawback in that the generosity makes it easier for coercions to compete. Therefore more combinations of coercions will be incoherent. When using a syntax-based matching, if one wants a particular pair of convertible (but syntactically different) types to be matched, then this can be achieved with a little extra work by defining an identity-like coercion between the two types that has no computational effect other than to change the type of its argument. This gives the user a finer degree of control, but they have to put in extra effort to achieve it. For example, a defined identifier is convertible with its definiens, but is not a syntactic match for it. To allow this match requires the user to define a coercion explicitly. One might wonder why this match could not be accepted by default.

The reason is that it happens in practice that one wishes to define combinations of coercions that would be coherent using syntax-based matching, but that become incoherent when using a more generous matching relation. This occurs since one may wish to use representations of distinct abstract concepts that are nevertheless convertible within the framework of type theory. One example might be using lists to represent both polynomials and finite sets. One would then have the ability to use a common existing library of list operations and results to define further constructions over both these types. However, one would not want the coercions defined on polynomials to be synthesised for finite sets, and vice versa. Under a syntax-based form of matching, such syntheses can be prevented

by giving the representations different names. Another example occurs later in this chapter (page 108.)

Ignoring the representation of some abstract type in some circumstances but accessing it in others can be seen as an information hiding problem. This probably has a more general solution borrowed from the realm of object-oriented programming, but that is not my concern in the current implementations I consider here.

A fourth reason for the different approaches is that in the case of the LEGO system there is already a specific algorithm available for computing the principal types necessary to perform a syntactic match, and its operation is familiar to the user. In the more theory-based presentations of Luo and Barthes, formalising the notion of principal types required for syntax-based matching would add extra baggage to the existing elegant theories.

For the above reasons, syntax-based matching seems preferable to conversion in present implementations. However, I do note that my decision on this was reached in tandem with other decisions (*e.g.* the wish to allow parameterisation.) Also, there may be arguments for using some relation less generous than full convertibility, but more so than syntax-based matching.

5.2.5 Graph shape

The simplest restriction that might be placed on the shape of the coercive graph is that there should be only one path between any pair of types. Forests and some other graphs would satisfy this restriction. As well as avoiding any difficult decisions about resolving competition between paths, it would make any dynamic graph-searching algorithm much simpler to implement.

However, once again a simple approach has the disadvantage of reducing flexibility and expressiveness. It would prevent two varieties of coercion combinations which I have found to be common and useful in practice. These are depicted in

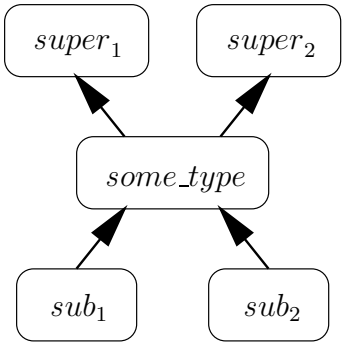

Figure 5.1: Graph without sharing

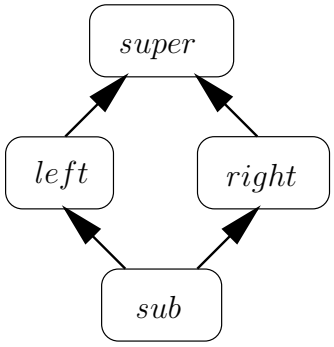

Figure 5.2: Graph with sharing

figures 5.2 and 5.3.

The restriction does not force a forest structure on the graph; some non-treelike pieces of connected graph have no competing paths, as can be seen by Figure 5.1. The type *some_type* can be coerced to both $super_1$ and $super_2$, and both sub_1 and sub_2 can be coerced to *some_type*. However, it does prevent what might be described as supertypes with sharing. This occurs when two types $left$ and $right$ to which some subtype *sub* can be coerced themselves have a common supertype *super*, as shown in Figure 5.2. In such a case there are two competing paths in the graph between the types *sub* and *super*; one via the type $left$ and the other via the type $right$.

I should give an example of supertypes with sharing in practice. *super* might be the type of maps. Let $left$ be the type of injections and $right$ be the type of surjections. It is clear that coercions from these two types to maps would be useful. Now if *sub* is the type of bijections it would seem useful to be able to coerce a bijection to be either an injection or a surjection. These coercions would produce the competing paths to the shared map supertype shown in the diagram.

Another example might concern flavours of algebras. For example, suppose that *super* is the type of sets, $left$ that of discrete sets, $right$ that of groups, and *sub* that of discrete groups. Again, all the coercions in the graph would be useful in practice but there are competing coercions from discrete groups to their

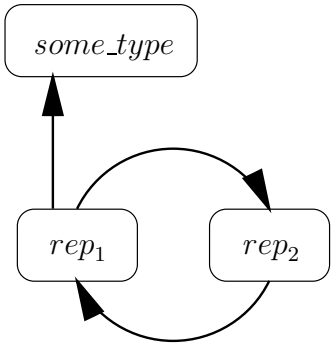

Figure 5.3: Graph with cycle

underlying carrier sets.

The second circumstance occurs when one wishes to use coercions to move between two alternative representations of the same concept so that functions can be applied to such objects without having to take into account on which representation the functions were originally defined. Such coercions introduce a cycle into the graph, as can be seen in Figure 5.3. Even if coercion definitions are not allowed to be used more than once within a path, cycles still cause competing coercions; for example, there are two paths from rep_1 to *some_type*; one involving the single edge between them, and another which first loops around the cycle via rep_2.

This is the circumstance that occurs when one has two different but essentially isomorphic concrete representations within the type theory of the same abstract concept. For example, a finite sequence could be represented as a list (rep_1) or as a tuple (rep_2.) In some circumstances the list representation might be the most convenient to use; in others the alternative tuple representation might be preferable. A coercive graph like that of Figure 5.3 would allow the use of both these representations interchangeably.

The approach taken within LEGOwcs is to allow sharing, but to insist that all competing paths are coherent. The different possible paths thus produce the same result after they are applied. In the case of a graph with a cycle, the

graph will remain coherent so long as the application of any cyclic path has no net effect. This concurs with the intuition that rep_1 and rep_2 are isomorphic representations, and that the coercions between them encode this isomorphism.

5.2.6 Other approaches to coherency

There are other choices that could have been made in approaching the matter of coherency between competing coercions. I explore some of them in this section.

5.2.6.1 Allowing incoherent coercions

One different approach that could be taken would be to allow incoherent competing paths, but to have some deterministic means by which to decide which should be synthesised. In the Coq implementation, if two different paths could be synthesised, then the path that was generated first is the one that is used. This means that one must consider the latest coercion definitions that contributed to each competing path, and then choose the path for which the latest definition is the older. Thus the coercive graph is determined by an ordered sequence of coercion definitions, rather than by an unordered set.

Allowing and resolving incoherencies gives the user more freedom, since, when writing a formal development, they can decide to define coercions which compete in ways that are not necessarily coherent, so long as they can keep track of which will actually be synthesised in a given circumstance. For someone familiar with the system this may not be an unreasonable responsibility.

However, I did not consider it to be an acceptable compromise for LEGOwcs, since the system was developed in order to write mathematics in a way that is as close as possible to informal practice without introducing any ambiguities. The exact formal structure of a coercive graph would be difficult for someone without experience of the particular type theory in question to keep in mind whilst they

read a formalisation. It seemed preferable to prevent any dangerous ambiguities by insisting on coherency, and thus allowing the user to rely on their intuitions as to the meaning of the implicit syntax.

5.2.6.2 Dynamic coherency checking

Another alternative to insisting on coherency would be to allow any combination of coercion definitions, but to consider a term to be ill-typed if incoherent competing coercions could be synthesised within it. This would act as a reasonable compromise between the two approaches outlined previously. It would allow the user to define coercions that generated potential incoherent competitors, but would prevent them from actually introducing any such ambiguities into the expressions they write. Alternatively, perhaps the system should only allow incoherent coercions to be defined if the user gives some explicit indication as to how to resolve the ambiguities that could arise as part of the definition. The important motivation for me is to ensure that the reader of a formal presentation cannot be misled into assuming that one coercion is being synthesised when in actuality a different incoherent coercion is synthesised by the system.

It was pointed out to the author by Zhaohui Luo that although the coercive graph may contain an infinite number of paths, only finitely many are ever synthesised within the course of a given formalisation. Since synthesised coercions are simple, unparameterised terms of the type theory, this reintroduces the possibility of checking the coherency of individual paths directly rather than working with parameterised paths. However such an approach would mean that type-checking would become slower and slower as the length of the development, and hence the number of coercions synthesised, increased. Still, in an efficient enough implementation, this approach would remove the need to statically check the coherency of the graph. (The paths themselves might still be statically generated,

in order to maximise the run-time efficiency of the system.)

5.2.6.3 Matching using conversion

Another ambitious change would be to attempt to allow types to be matched by some flavour of conversion rather than a syntax-based comparison, although as previously noted, synthesis within such a system would still necessarily be incomplete. Still, the argument synthesis used in existing proof-checkers is incomplete, as in its full generality it requires higher-order unification; nonetheless it has been found to be very useful even in its restricted form. If the additional problems of checking coherency requirements and resolving ambiguities were dealt with by some other means (such as those outlined in the previous two paragraphs) then an incomplete but more extensive system of coercion synthesis might also be found to be useful in practice.

5.2.6.4 A problematic combination of coercion synthesis and argument synthesis

In LEGOwcs, newly generated coherent competing paths are ignored since their competitors are assumed sufficient for synthesis. In fact, in certain interactions with the LEGO argument synthesis mechanism, the nodes via which a synthesised coercion runs can determine whether or not the term in which it is synthesised can be made to type-check. A final option that might be considered would be to store all the coherent competing paths between two types and to try instantiating the later paths if using the earlier ones did not allow the whole term in question to type-check. Since coherent coercions have the same computational effect, nothing should be lost by remembering only one of them. But in fact there is a circumstance where the direction the path in the graph takes is important, and I shall describe it below.

Recall the situation presented in figure 5.2. Suppose now that the path between *sub* and *super* that is generated first goes via the type *right*. Then the coherent path generated later that goes via the type $left$ would not be added to the graph. Suppose now that elements of *super* can themselves be considered to be types. A concrete example of this situation might be that presented on page 97, where *super* is the type of sets, which can be coerced to their element types. In this example $left$ is the type of discrete sets, *right* the type of groups, and *sub* the type of discrete groups.

Now suppose that $T : \{\Pi S \,|\, \mathsf{discrete\_set}\}\ S \to \ldots$ is a construction defined on the elements of discrete sets. Let G be a discrete group, and let x be an element of type G. Then one cannot apply the operator T to x, because the necessary first argument, G considered as a discrete set, will not be synthesised, since the type of x is G considered as a group, considered as a set. However if the paths had been generated in the other order, then the type of x would be G considered as a discrete set, considered as a set, and so the application of T to x would be successful.

This is a reasonably subtle point, and it occurs rarely enough in practice that I did not consider the extra computation that would be required to try all possible combinations of coherent coercion syntheses before failing to type-check a term to be worthwhile for my own purposes. In the current implementation, the problem can be avoided by the writer choosing an appropriate order in which to make the coercion definitions; a reader need not be concerned by it.

5.3 Example material

Although the remaining points I wish to make about coercions in this chapter may be considered at the level of abstract examples, in order to motivate them and to illustrate them clearly, it is useful to have a concrete example from which to

work. The best such examples are provided in the context of a general framework for mathematics. Such a framework will be developed later in this thesis for the Galois case-study, within chapter 8; however, its presentation benefits from the understanding of material in preceding chapters, and parts of these (*e.g.* section 6.1.4.2) in turn are best read once this chapter has been understood.

In order to break this unfortunate cycle of dependencies, I instead introduce a simple concrete example at this stage. It is independent of the case-study but may be understood without further preparation. Some further discussion and some richer illustrations of concepts discussed in this chapter (particularly those to do with overloading), may be found in sections 6.1.4.2 and 8.6.

I define the type of natural numbers, nat, by induction, using the standard constructors of zero, 0, and successor, +1.

▷ $\mathsf{nat} : \mathsf{Type}_0$

▷ $0 : \mathsf{nat}$

▷ Allow +1 to be written postfix

▷ $+1 : \mathsf{nat} \to \mathsf{nat}$

An equality relation is defined upon nat by recursion.

▷ $0 = 0 \simeq \mathsf{true}$

▷ $m{+}1 = 0 \simeq \mathsf{false}$

▷ $0 = n{+}1 \simeq \mathsf{false}$

▷ $m{+}1 = n{+}1 \simeq m = n$

On this type I recursively define a doubling operator, doub_n.

▷ $\mathsf{doub}_n : \mathsf{nat} \to \mathsf{nat}$

▷ $\mathsf{doub}_n\ 0 \simeq 0$

▷ $\mathsf{doub}_n\ n{+}1 \simeq (\mathsf{doub}_n\ n){+}1{+}1$

I now introduce a type representing the even numbers. These will be natural numbers that satisfy a certain predicate; that they are equal to some other natural

number doubled.

▷ Define $\mathsf{is\_even} = [\lambda n : \mathsf{nat}]\ \langle \exists m : \mathsf{nat} \rangle\ (\mathsf{doub}_n\ m) = n : \mathsf{nat} \to \mathsf{prop}$

An even number will be represented by a natural number coupled with a proof of its evenness. A projection coercion allows all even numbers to be considered as natural numbers.

▷ Define $\mathsf{even} = \langle \Sigma n : \mathsf{nat} \rangle\ n.\mathsf{is\_even} : \mathsf{Type}_0$

▷ Define coercion $\kappa_1 = [\lambda n : \mathsf{even}]\ n.\mathbf{1} : \mathsf{even} \to \mathsf{nat}$

There is an alternative definition for the type of even numbers, by induction, resulting in a type isomorphic to nat:

▷ $\mathsf{even'} : \mathsf{Type}_0$

▷ $0_e : \mathsf{even'}$

▷ Allow +2 to be written postfix

▷ $+2 : \mathsf{even'} \to \mathsf{even'}$

A recursively defined computation coerces between even' and nat.

▷ $\kappa_2 : \mathsf{even'} \to \mathsf{nat}$

▷ $\kappa_2\ 0_e \simeq 0$

▷ $\kappa_2\ n{+}2 \simeq (\kappa_2\ n){+}1{+}1$

5.4 Connections with subtyping

Work on coercions in type theories developed partly from attempts to extend theories with the ability to allow some classes of object to inherit properties and features from other classes. In the language of object-oriented programming, one would say that child classes inherit from their parent classes, or that subclasses inherit from their superclasses. Coercions also allow an object to act as if it had more than one type. Both these notions are related to the paradigm of *subtyping*. The extension of type theories with subtyping is a common idea in the literature of the subject. One of the uses of coercions is that they can give a computational

explanation for subtyping.

The basic principle of subtyping is that if S is a subtype of T, then terms of type S may be considered to have type T as well. This allows a term with a subterm of type T to remain well-typed when a second subterm which has type S is used in place of the original.

One form of subtyping that has received a fair amount of investigation is subtyping based on records[BT95]. In this approach, objects are tuples (or records) whose components correspond to named fields. One record type, S, is a subtype of another, T, if S contains all the fields of T (and possibly some additional ones too.) Whenever a term with type S is used, one can simply forget the components in the extra fields to retrieve a term with the record type T. There is thus some computational way in which to give a meaning to this form of subtyping.

Coercions extend the idea that subtyping and other forms of inheritance can be justified computationally[BCGS91] by allowing functions of a more general nature than "forgetful" ones that simply get rid of information held in extraneous fields of a record. But a fair number of the coercions in my case-study development are forgetful ones. As another example of the connection, the Coq system including coercions allows records to be defined and used in the sort of style commonly seen in systems with record subtyping, and it implements this through the definition of an appropriate collection of coercions.

5.5 Connections with overloading

Coercions can be seen as implementing a form of overloading. In an overloading system, a single lexical phrase may represent several different things, and the meaning intended is recovered by examining the context within which the phrase is used.

The context is usually provided by a function application, where the function

(or argument) provides some contextual information that allows us to resolve the meaning of an overloaded argument (or function.) In section 5.1 I observed that if coercions are considered at the level of a logical framework then all context-providing circumstances (such as using an expression as a type or as a function) may be seen as the application of a function (a type-theoretic operator) to an overloaded argument.

When using a coercion, all expressions representing objects whose computed types lie within a certain class are effectively overloaded. Coercions thus implement a general pattern of *typed* overloadings.

5.5.1 Overloading of individual identifiers

Another method of overloading is to provide meanings only to individual phrases (normally identifiers) rather than to all of a typed class. Using a simple trick, coercions can be used to implement this form of overloading also. However, the system implemented by LEGOwcs would need to be modified in order to make this technique more useful and flexible. I believe there is fertile ground to be explored here; unfortunately, the possibility arose too late in my research to allow me to pursue it properly. I present here only an outline of the technique and some thoughts on how it might be developed.

5.5.1.1 The desired overloading

We know that *all* even numbers can be considered as natural numbers, and have encoded a computation that coerces them in this way. However, conversely we know that *some* natural numbers can be considered as even numbers. For example, consider the number 4. This has representations both as a natural number,

▷ Define $4_n = 0{+}1{+}1{+}1{+}1 : \mathsf{nat}$

and, because I can prove that 4_n is equal to doub_n applied to the natural number

two (0+1+1), it also has a representation as an even number.

▷ P : $(\mathsf{doub}_n\ 0{+}1{+}1) = 4_n$

▷ Define $4_e = (4_n, 0{+}1{+}1, P : \mathsf{even})$: even

I would like to be able to use the same identifier to represent both these objects; four in its form as both an even and as a natural number. One way to do this would be to define the general identifier to be equal to the even form:

▷ Define $4 = 4_e$: even

This would allow me to use 4 as a natural number through an implicit synthesis of the coercion κ_1, since

▷ $\kappa_1\ 4 \simeq 4_n$

This solution is used often in the case-study. However, it is not optimal for several reasons. The first is that I was forced to introduce a subsidiary identifier, 4_n, in order to refer to 4 as a natural number before I was able to introduce its "real" name. The intervening uses of 4_n (for example, those in the definition of $4 = 4_e$ itself) are rather anomalous.

It also has a second disadvantage; I have to know and limit the shape of the coercive graph in advance, for I must define the identifier to have the type at the source-most end of the coercive path if I am to use it for all the other superclasses. For example, if I wish later to introduce a new type of multiples of four, from which a coercion to even numbers could be defined, I would not be able to "extend" the overloading of 4 to have this type.

Another problem also concerns the shape of the coercive graph. Perhaps I would like for 4 to represent an element within the type of square numbers also. Now there is no source from which all nodes that represent types I would like to overload 4 to have are descendants. I could introduce a special type of even square numbers especially for this purpose, and define 4 as an element of this type; but this is clearly an undesirable inconvenience.

Instead, I should like to overload the *identifier* 4 and let it be overloaded to represent objects of arbitrary types. I do not want to have to specify all these types in advance. However, I do want to ensure that the overloading is coherent with other abbreviations used.

▷ Forget back through 4

5.5.1.2 Coercions from unit types

The goals set out above can be met through the use of coercions from unit types. I can form an inductively defined type $\top$ with one constructor $\star$. If necessary, I could define many isomorphic but non-convertible copies of this type; but in the existing LEGOwcs system I can just give the same type as many names as I want, and the system will consider them distinct for the purposes of synthesising coercions. This is an example of the sort of circumstance described in section 5.2.4 where matching at a syntactic (conceptual) level is preferable to matching at the conversion (implementational) level.

Thus, I define

▷ Define 4_is_overloaded $= \top$ **:** Type_0

▷ Define $4 = \star$ **:** 4_is_overloaded

Thus 4_is_overloaded is a type with a single inhabitant, the identifier 4. Through coercions, 4_is_overloaded will be used as if it were the type of all objects that can be represented by this identifier.

I can now define coercions to turn instances of this identifier into whatever types I choose. I shall use the values already defined as 4_n and 4_e, but I do not need to use these intermediate names or to define all the overloaded values at the same time.

▷ Define coercion $\kappa_3 = [\lambda\_ : 4\_\mathsf{is\_overloaded}]\ 0{+}1{+}1{+}1{+}1$

: 4_is_overloaded $\rightarrow$ nat

▷ P **:** $(\mathsf{doub}_n\ 0{+}1{+}1) = 4$

▷ Define coercion $\kappa_4 = [\lambda\_ : 4\_\mathsf{is\_overloaded}]\ (4, 0{+}1{+}1, P : \mathsf{even})$

: $4\_\mathsf{is\_overloaded} \rightarrow \mathsf{even}$

▷ $(4, 4)$ **:** $\mathsf{nat}\ \#\ \mathsf{even}$

Note that in the above 4 is used to represent 4_n in the coercion κ_4 corresponding to the definition of 4_e.

The graph produced by these coercions contains competing paths; there are now two paths from $4\_\mathsf{is\_overloaded}$, the general type of the identifier 4, to nat. However, these paths, κ_3 and $\kappa_1 \circ \kappa_4$, are coherent. Therefore no ambiguities have been introduced.

▷ $\kappa_3\ 4 \simeq \kappa_1\ (\kappa_4\ 4)$

Thus by using the existing coercion system for this new form of overloading I ensure that our previous coercive abbreviations and overloadings all cohere.

One could go on to define further coercions to future types of multiples of four or square numbers if so desired. For example, I can define a coercion that overloads 4 to have the other type representing the even numbers, even':

▷ Define coercion $\kappa_5 = [\lambda\_ : 4\_\mathsf{is\_overloaded}]\ 0_e{+}2{+}2$ **:** $4\_\mathsf{is\_overloaded} \rightarrow \mathsf{even'}$

5.5.1.3 Extending this technique

I would like to overload more complicated expressions than simple identifiers. For example, having proved that = is reflexive,

▷ Introduce R **:** $\{\forall n : \mathsf{nat}\}\ n = n$

it is easy to prove that the double of any natural number is an even number (from the definition of the is_even predicate.) Thus I should like to define a version of the doubling function that returns an overloaded result that can represent either an even or a natural number, depending on the context. As before, although this can be achieved by implementing the function that returns an even number and relying on paths from this result type, this solution would not later allow us to

extend the overloading of the function to *e.g.* return a multiple of four whenever it was applied to an even number.

By generalising a coercion definition so that it goes from a parameterised class of unit types, the previous technique can be extended to accomplish this:

▷ Define $\mathsf{doub\_is\_overloaded} = [\lambda\_ : \mathsf{nat}]\ \top\ \mathbf{:}\ \mathsf{nat} \to \mathsf{Type}_0$

▷ Define $\mathsf{doub} = [\lambda\_ : \mathsf{nat}]\ \star\ \mathbf{:}\ \{\Pi n : \mathsf{nat}\}\ \mathsf{doub\_is\_overloaded}\ n$

I can now define two coercions to implement the overloading as before.

▷ Define coercion $\kappa_6 = [\lambda n \,|\, \mathsf{nat}]\ [\lambda\_ : \mathsf{doub\_is\_overloaded}\ n]\ \mathsf{doub}_n\ n$
$\mathbf{:}\ \{\Pi n \,|\, \mathsf{nat}\}\ (\mathsf{doub\_is\_overloaded}\ n) \to \mathsf{nat}$

▷ Define coercion $\kappa_7 = [\lambda n \,|\, \mathsf{nat}]\ [\lambda\_ : \mathsf{doub\_is\_overloaded}\ n]$
$(\mathsf{doub}\ n, n, R\ (\mathsf{doub}\ n) : \mathsf{even})\ \mathbf{:}\ \{\Pi n \,|\, \mathsf{nat}\}\ (\mathsf{doub\_is\_overloaded}\ n) \to \mathsf{even}$

▷ $(\mathsf{doub}\ n, \mathsf{doub}\ n)\ \mathbf{:}\ \mathsf{nat}\ \#\ \mathsf{even}$

Again, this notation is known to be unambiguous because LEGOwcs has automatically checked that two competing coercions from $\mathsf{doub}\ n$ to nat, κ_6 and $\kappa_1 \circ \kappa_7$, are coherent:

▷ $\kappa_6\ (\mathsf{doub}\ n) \simeq \kappa_1\ (\kappa_7\ (\mathsf{doub}\ n))$

5.5.1.4 Problem cases

So far, so good. However, now consider an attempt to implement this overloading for the other definition of even numbers, even'. I can define an appropriate function by recursion:

▷ $\kappa_8\ (\mathsf{doub}\ 0) \simeq 0_e$

▷ $\kappa_8\ (\mathsf{doub}\ n{+}1) \simeq (\kappa_8\ (\mathsf{doub}\ n)){+}2$

However, I cannot define this function as a coercion. The LEGOwcs system reports that incoherent paths have been generated. This is because the coercions generated as κ_6 and as $\kappa_2 \circ \kappa_8$, both from $\mathsf{doub}\ n$ to nat, do not produce results that LEGOwcs can see are convertible for arbitrarily-valued instances of the parameter n. In fact, such terms *are* convertible, and this can be proved by

induction on n; but LEGOwcs cannot perform such meta-theoretic reasoning.

In fact, a similar problem prevents me from defining a cycle in the graph that equates the two representations of the type of even numbers, even and even'. Again, LEGOwcs will fail to find the cyclic paths coherent, although it will be provable by induction that they are.

This sort of problem suggests that an ability to *prove* some of the conversion relationships required for coherence might be a useful addition to the coercion system. I have not pursued this possibility, but I suspect that it would be particularly suitable for a treatment of coercions at the level of a logical framework, where equality (conversion) judgements can be derived by other reasoning in the framework.

There is another way in which the current framework does not provide the exact features that are needed to implement a completely general form of overloading through coercions. As it stands, all arguments to doub must be subtypes of nat. Ideally, one would be able to define doub itself as an overloaded but unparameterised identifier that could take arbitrary arguments and have any arity. In order to do this, one would need to have coercions from the doub_is_overloaded unit type to several different functional types, and have the correct coercion synthesised in response to the application of the non-functional doub identifier. The present system allows only one Π-coercion to be defined on each type. As mentioned earlier, this feature of the implementation is only absent because I saw no need for it at the time of programming and wished to keep things as simple as possible rather than because there is some fundamental reason why this cannot be done.

5.5.2 Coercions and type classes

Mechanisms for inheritance and associated overloadings are also provided by another paradigm, that of type classes. Simple type classes are most well-known through their use in functional programming languages such as Haskell. Such type classes and languages deal only with computation, but an extension of the paradigm, so-called axiomatic type classes, makes it suitable for type-checking where logical properties and proof are relevant. The Isabelle proof-checker[Pau93] is one that uses axiomatic type classes[Wen97].

I present a very quick review of this paradigm and how it works relative to coercions. I should warn the reader that my understanding of type classes is not as complete as I would like, and I have little experience of using them in practice. Nevertheless perhaps drawing some analogies between the two approaches may be revealing.

I shall illustrate some type class concepts using some Haskell-like pseudocode. An example of a type class might be one of equality types. These would be defined to be types that had an associated equality function.

```
class Eq a where
 (==) : a -> a -> Bool
```

A subclass of equality types that also had an order defined on them could be specified.

```
class (Eq a) => Ord a where
 (<=) : a -> a -> Bool
```

A type class definition `C a` can be seen as corresponding to the definition of a record type `C` with the type `a` as its first component, with appropriate subtyping and inheritance functionality being defined. (Thus functions on types in the class

`Eq` above are inherited by those in the class `Ord`. One can see this as a projection coercion from `Ord` to `Eq`)

Given a particular type (*e.g.* the built-in `Bool`) and the definition of a particular function `iff` with the appropriate type,

```
iff true  true  = true
iff true  false = false
iff false true  = false
iff false false = true
```

one can place the type `Bool` as an *instance* of the class `Eq` as follows:

```
instance Eq Bool where
 x == y = iff x y
```

Note that a similar kind of instance declaration through coercions from unit types was explored in the previous section. With type classes it is the most basic notion.

As it stands, there is no information about the logical properties of these types. So, if one had defined some unsuitable `dumb_equality` function,

```
dumb_equality : Bool -> Bool -> Bool
dumb_equality x x = false
```

then this would be equally acceptable as an instance of `Eq`. In order to express logical properties of type classes one needs an *axiomatic* extension to the paradigm. (Such an extension is not native to Haskell, but I shall continue to use the same notation.) Thus one could specify instead

```
class Eq a where
 (==) : a -> a -> Bool
 Reflexivity  :     {x : a} (x == x)
 Symmetry     :   {x,y : a} (x == y) -> (y == x)
 Transitivity : {x,y,z : a} (x == y & y == z) -> (x == z)
```

Now the equality function is guaranteed to have the intended properties. An instance definition `Eq Bool` using `iff` for equality could provide the required proofs, but one using `dumb_equality` could not.

For example purposes, I also define a couple of other axiomatic type classes for preorders and orders.

```
class PreOrd a where
 (<=) : a -> a -> Bool
 Reflexivity  :     {x : a} (x <= x)
 Transitivity : {x,y,z : a} (x <= y & y <= z) -> (x <= z)


class (Eq a) => Ord a where
 (<=) : a -> a -> Bool
 Reflexivity  :     {x : a} (x <= x)
 Antisymmetry :   {x,y : a} (x <= y & y <= x) -> (x == y)
 Transitivity : {x,y,z : a} (x <= y & y <= z) -> (x <= z)
```

These sorts of extensions seem natural enough in a dependently-typed language that contains propositions and proofs such as LEGO, and indeed they have been implemented in Isabelle.

As well as concrete instantiations of type classes (such as `Eq Bool` above), one can also define abstract instances. (The proofs of the axioms have been omitted since I don't want to have to introduce further notation to deal with them.)

```
instance (PreOrd a) => Eq a where
 x == y = (x <= y & y <= x)
 Reflexivity = ...
 Symmetry = ...
 Transitivity = ...
```

Abstract instances can be seen as analagous to general coercion definitions; here a coercion has been defined between `PreOrd` and `Eq`. I note some symmetry here, in that general coercions can be used as a basis for defining concrete instances, but in type classes the basic notion of a concrete instance is instead generalised; the two approaches seem to share a similar functionality, but to take different ideas as their fundamental building blocks.

There remain some differences between the approaches, however. Coherence is effectively for free with type classes. This is pleasant in some ways but is also a result of there being some limitations on the inheritance graphs that can be built using the approach.

Also each type `a` can be considered as an instance of a particular class `C` in only one way. Thus one could not define many different pre-orders on any one underlying type and then refer to them by different names. This can probably be avoided through the syntactic trick of duplicating definitions, however. Again I note some interesting symmetry in that this same trick is effective in avoiding coherence problems with coercions.

Finally, type classes are by definition concerned with types. Some other uses of coercions, such as implementing other syntax tricks of abbreviation (*e.g.* overloading a natural number to represent a set), cannot be implemented through type classes.

Further work on exploring the differences and similarities between these two approaches would probably be very worthwhile.

Chapter 6

Proof style

This chapter looks at the decisions that have to be made about the overall style of proof one will use when working in a formal system. The issues are ones of presentation; other important foundational notions to do with content are considered in chapter 8. For a literate presentation both style and content are important.

I review some of the major practices of the traditional informal style that aid readability and consider how to translate them to produce a literate formalisation. Some issues specific to the LEGO proof-checker and the particular case-study on which I worked are also considered. The proof style I chose for the presentation of the case-study is explained.

6.1 Informal mathematical practice

This section reviews some of the practices that are common in writing informally about mathematics and that distinguish it from more formal styles of presentation. Each subsection describes such a practice, considers how it might be used in a formal framework, and concludes by outlining how the issue was handled in the case-study development.

6.1.1 Use of natural language

Formal mathematics involves the manipulation of strings of symbols. The expressions formed represent other concepts, but they tend to be manipulated by considering syntax-based rules rather than the semantics of what is being represented. (The boundary between syntax and semantics cannot be precisely defined, but a distinction is often usefully drawn in practice.)

Informal mathematics makes some use of syntactical reasoning also. We may manipulate symbolic expressions such as equations through rules which have previously either been agreed upon as axiomatic or proved to be appropriate. But such expressions tend to form only a part of a mathematical development. Often much of the mathematics is expressed in natural language. This change in style between natural language to expressions provides a useful visul clue that aids the reader in understanding the structure of the development.

Very broadly, natural language tends to be used in circumstances similar to the following:

- when the identifiers used to describe objects have become whole words and phrases. (Most identifiers used in expressions are very short.)
- to indicate logical operators such as implication and quantification in both the statement of results, and in the structure of the proofs of these statements. (Phrasings such as "for all ...", "if ... then ...", "Therefore ..." and "Suppose ..." are examples.)
- to indicate clues to the higher-level structure of a proof, so that the reader can intuit missing details. (Examples would include phrasings such as "similarly", or even the well-loved "trivially.")

The grammar of natural language is far more complex than that governing most expressions. There are many idioms and synonyms, which makes for more

interesting reading. In some ways natural language can be verbose and redundant; many phrasings for simple concepts are very long compared to their equivalents in symbolic expressions. It is also very linear; expressions can make much better use of spatial juxtaposition. But the complex grammar also allows for better use of analogy and the leaving implicit of information that must be explicitly supplied to satisfy the simpler grammar of expressions. These practices sometimes risk ambiguity, but they also make it possible to present only the information most relevant to understanding.

Because informal mathematics uses natural language, it tends to require less use of names and labels than does its formal counterpart. Proofs and other objects that are constructed only for immediately use can be left anonymous and referred to by pronouns such as "it" and "this". Especially when working in a classical logic which admits proof irrelevancy, most proofs do not have to be named at all; instead, later proof-steps in a chain of reasoning are understood to be immediate consequences of earlier ones. This can be indicated by phrasings such as "Thus...", "So..." and "Therefore..."

In the case-study, I work in a fully formal framework, but with the ability to add informal commentary in natural language where it seems appropriate. At some levels of presentation, this commentary replaces omitted formal statements; the full formal development can be consulted if a reader wants all the details.

I experimented with using long identifiers that mimicked natural language phrasings for objects in earlier versions of the development. However, I ultimately found them to be too verbose, and to compromise one of the few advantages of expressions: that their structure is evident from syntax. Further, the language that such expressions offered was sometimes a little *un*natural and awkward to read. In the final version, I tend to use shorter, more symbolic names in most places. though the names of predicates do often have an "is" prefix, and constructs such

as "as a subset of" still survive.)

I made some effort to structure the proofs using natural language phrases such as "Introduce...", "Let" and "Suppose ... and ..." This worked reasonably well, although a lack of synonyms lead to some rather repetitive phrasing.

I didn't try to solve the problem of anonymous identifiers, so the development is afflicted by a preponderance of identifiers that are defined to be used once only in the following definition.

6.1.2 Expressions

In informal mathematics, symbolic expressions are used when we consider complicated combinations of operators, especially when there is a useful correspondence between the semantics and syntactic rewriting rules. Expressions tend to be succinct, using short identifier names, and can make great use of spatial juxtaposition. Both dimensions of the page can be used through subscripts, superscripts, and symbol stacking. Conventions for the associativity and fixity of operators are also used to make expressions more readable.

Sometimes other implicit conventions mean that expressions carry some proof information. For example, we might write "$x \approx y \approx z$" to mean "$x \approx y$, and $y \approx z$, so $x \approx z$ using a previous proof or assumption that $\approx$ is transitive." A chain of equational reasoning may also be annotated with indications of the rules or axioms justifying each step.

In the case-study, everything is necessarily an expression to some degree. For built-in operators (such as those controlling the direction of the proof, and abstraction over variables) I tailored the output to help readability using arbitrary fixity conventions, and added annotations in the proof to flag certain objects as propositional so they could be rendered differently. I also made use of natural language constructs where it seemed appropriate.

For identifiers defined by the proof script, I provided rather less control. I allowed some fixity declarations, but nothing as complicated as the sort of "around-fix" notation used for operators such as integrals in everyday mathematics. Nor were there any notions of default associativity. Also, arbitrary subscripts and superscripts would have pushed my rather ad hoc system of automating line-breaks using simple LaTeX past breaking point, so I didn't allow these either.

This made my expressions somewhat linear and one-dimensional. These restrictions resulted only from lack of time in which to implement something more complex, not because of any essential problems with pretty-printing. In the future development of systems for literate formalisation, I would recommend allowing pretty-printing to be defined to follow a much more flexible grammar.

6.1.3 Naming conventions

As already mentioned, the identifiers used in expressions tend to be short, with names such as "G" and symbols such as "+". Longer identifiers tend to be used for concepts at a slightly higher level of abstraction, where they can be swaddled in natural language constructs. ("Suppose G is a *group*", "Since $+$ is *associative*") Both in and out of expressions, informal mathematics makes use of a wide range of symbols and different fonts in order to increase the available space of short names. Thus we might see "$\mathcal{G}$" and "$\oplus$" used also.

Sometimes a name carries implicit typing information. After a few uses, an author may cease to mention that G is a group. These conventions are common in mathematics. A reader knows that "$\approx$" is likely to be a binary relation that will be written infix, so sometimes an author will not mention this information at all, even though strictly speaking it is necessary to state the convention in order to avoid parsing problems.

Another quirk of naming in informal mathematics is that the same identifier

may be used to refer both to a statement and to a proof of that statement. So "Fermat's last theorem" may be introduced as an abbreviation for the proposition "For all $x, y, z, n > 0$, if $x^n + y^n = z^n$ then $n < 3$", but later the identifier will be used to refer to a *proof* of this theorem: "By Fermat's last theorem, we have that ..." This is reasonable since in applying the result one is not normally interested in the content of a particular proof of it, only the fact that some proof of the result exists. In type theory, where proofs are first-class citizens and there way be many different proofs of the same result, there is a need to give separate names to results and their proofs.

Only the most important of results tend to be given long names. Some others may be left anonymous, either because they are used immediately, as described earlier, or because they may be described without naming them ("By the transitivity of $\approx$...") The remaining relatively unimportant results and axioms tend to be identified with short labels, even if there are too many of them to keep track of.

Most readers of mathematics don't seem to mind having to refer backward a few pages to find out what a label refers to. Presumably a lot of mathematics would benefit from a hypertext presentation where the use of a term could link to its definition. This interesting idea is not explored by this thesis, as it would entail another paradigm shift; moving from an informal presentation to a literate formalisation is a big enough step in itself. However, I suspect that non-linear, hyperlinked presentations may become more and more common as mathematicians discover the benefits of this technology for electronic publication.

In the case-study, I allow identifiers to be associated with default types so that the typing information can be omitted in later uses. The pretty-printing system using pieces of LaTeX for identifiers makes the presentation many times easier to read than using the ASCII identifiers of LEGO; having a large vocabulary

of symbols is a great aid to literateness. When I wished to name both results and their proofs, I tended to use lower-case for results and upper-case for proofs, but mostly I did not have to give names to propositions. For proof identifiers, I tended to use short lower-case descriptions for axioms and lemmas to be used in equational proofs (such as "plus_assoc" for the result that might informally be called "the associativity of +") and short indexed capitalised labels for more substantial proofs (such as "GROUP_1" or "FINDIM_2".)

6.1.4 Overloading

Overloading occurs when an identifier may have distinct meanings in different contexts. Some of the ways in which this device is used in informal mathematics can be quite subtle.

6.1.4.1 Overloading through scoped namespaces

Sometimes the same identifier will be used for essentially unrelated objects. For example, the proof/result "Lemma 1" might be defined in many different ways even in a single book. Across the entire literature of mathematics, generic-sounding adjectives such as "good" or "strong" are defined and used in many dissimilar senses.

To a certain extent, the same could be said of other common terms such as "normal", "0", and "continuous". But there *is* some analogy between the different uses of the same term in these cases, and I wish to leave considering this until subsection 6.1.4.3. However, which use is actually intended may be decided partly in a similar manner to the way in which we resolve the use of such identifiers as "Lemma 1" above.

The reader of an informal text resolves these sorts of ambiguities partially through looking at how they are being used, as we do with homonyms in English.

But mostly the resolution relies on some notion of the locality of the text in which they are found. If two chapters of a book prove different "Lemma 1"s, then the later use of "Lemma 1" in one of those chapters is interpreted as referring to the one that was proved in the same chapter. If "Lemma 1" is used outside these two chapters, it will probably be accompanied by a further qualifier that specifies in which chapter the result is found. The chapters define areas within which the use of the identifier is unambiguous.

Something similar occurs with wider areas such as whole books. One can even see different parts of mathematics as defining still wider areas within which uses of overloaded terms can be resolved. For example, "root" is likely to have different interpretations in analysis and in graph theory. Conversely, the use of the more common identifiers, such as those for variables like "x", "y" and "z", can be resolved by looking in more localised areas; subsections of a chapter, paragraphs, or even sentences.

Thus there seems to be an implicit notion of namespaces, where the meaning of an identifier can be ascertained at least in part through looking at the area in which it is used. This idea is used with rigour in defining the behaviour of computer programs, where a notion of "scope" is often fully formalised. In a programming language such as SML there is an explicit action of opening a module and its associated namespace. Whilst a namespace is open, it's identifiers are in scope and can be used unambiguously without further qualification. In a more general context when the module is closed, the identified items can be obtained by qualifying the basic identifier with the name of the module. The difference between uses such as `bar` and `foo.bar` is more formally defined than that between identifiers in mathematics such as "Theorem 2" and "Theorem 2 from *Ideal Examploids* by E. G. Maidup", but seems analagous to it.

Whilst this all follows clear common sense conventions, and presumably can

be formalised easily, it is as well to be aware of the way that simple namespace-manipulation occurs implicitly in mathematics. This is because there are other forms of overloading where different matters arise either instead of or in addition to the issue of namespaces, and where formalisation becomes more difficult.

6.1.4.2 Simple overloading through coercions

Sometimes, there is a relatively simple relationship between the different objects that can be represented by an overloaded identifier. Coercions can be seen as implementing such a form of overloading, in which immediate contextual clues are used to resolve the ambiguity, as described in chapters 4 and 5. In this case the framework within which this is performed is formal enough that the ambiguities can be resolved entirely by machine.

There are two ways of seeing such coercions as overloadings. One is that the identifiers representing the coerced objects have been overloaded so that they can stand for both the original object and the coerced object. This was the view taken in the implementation of coercions in LEGO. However, a second perspective is that the operator itself is being overloaded, rather than the arguments to which it is being applied.

Suppose we have a coercion $\kappa : \mathbb{Z} \to \mathbb{R}$ between the types of integers and real numbers, and also a square root operator $\surd : \mathbb{R} \to \mathbb{R}$ defined on the real numbers. If $n : \mathbb{Z}$ is an integer, then the expression $\surd n$ is well-defined, as it repesents $\surd\kappa(n)$. The first way of viewing this in terms of overloading is that n has been overloaded to represent also $\kappa(n)$. But one could also see it as overloading $\surd$ to work on integers by representing the composition $\kappa \circ \surd : \mathbb{Z} \to \mathbb{R}$.

This second perspective allows us to consider a related form of overloading that is not handled by the current system of coercions. Suppose we have a unary minus operator $\neg : \mathbb{R} \to \mathbb{R}$ which we would like to overload to work on integers

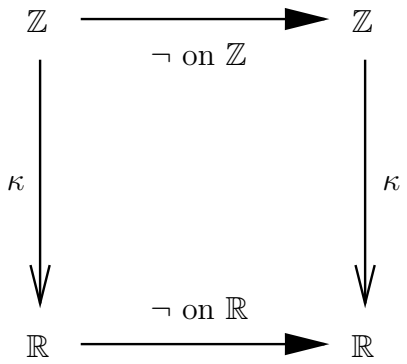

Figure 6.1: Coherency of ¬ and κ

also. In this case we want to change the result type of the operator as well as its argument type, so the first perspective will fail.

The overloading of ¬ in this way can be considered to be coherent with the coercion κ (*i.e.* it doesn't introduce unresolvable ambiguities) if the diagram show in figure 6.1 commutes.

In more simple cases, one can implement such an identifier overloading in the existing system using coercions from unit types, which will also machine-check such a combined condition of coherence. This possibility was discussed in subsection 5.5.1. However, further extensions to the coercion system would be necessary in order to make the use of this sort of idiom as flexible and as convenient as it should be. The possibility of extending the coercion/overloading system in this way came up too late in the day for me to pursue it, but I believe it would be a useful avenue to explore further.

6.1.4.3 Overloading through analogy

Informal mathematics often takes this method of overloading even further down the above route. Instead of just using the same identifier for two related concepts, sometimes one also uses the analogy between the two concepts implicitly and uses past experience and results proved with one in reasoning about the other.

The most simple example of this is when two different books define an object

in different but essentially equivalent ways. Let us take an abelian monoid as an example. A monoid is a set with an associative binary operator that has a left and right identity element. So we could specify an abelian monoid to be a monoid with an extra axiom that said the operator was commutative. But alternatively, we could define a new minimal collection of axioms for the abelian monoid: that the operator is commutative and associative, and has a left identity. We can do this since the right identity result follows from the left identity and commutativity axioms.

So we have two essentially equivalent definitions of an abelian monoid, but they aren't identical. Especially in a type-theoretic framework, they have rather different structures. However, we'd want to use them interchangeably in informal mathematics; results proved for one would be assumed for the other. We can see this as overloading the term "abelian monoid" to represent both definitions; but here we are combining the namespace type of overloading outlined in subsection 6.1.4.1 with an implicit appeal to an essential equivalence between the two definitions. Indeed, we would hope to be able to formalise this equivalence and prove that it lets us use the definitions interchangeably.

In fact, for the case-study one *does* have a formal framework that would allow us to do this; that of coercions. Every abelian monoid that fits the first definition could be turned into one that fits the second, and vice versa. Hence, one could use them interchangeably in a formalisation once we had defined these coercions.

However, in general, finding analogies between mathematical objects and their properties and formalising them in a common abstract framework is a lot of work. There are many very rich areas of mathematics that have grown out of such analogies, such as category theory, topology, and universal algebra; indeed, one could argue that pure mathematics *is* the study of high-level analogies. Informal reasoning makes good use of the more simple forms of such abstractions *implicitly*

to lessen the amount of detail that needs to be given. Within a formalisation, it is often desirable not just to do the work to set up the frameworks in the first place (which is clearly necessary to be sure of correctness) but also to be able to invoke these appropriate frameworks implicitly at suitable points. Having to be explicit with these invocations detracts from the readability of a development, but automating such subtleties is clearly a very big task.

In the case-study, I sometimes found it useful to define separate LEGO versions of an object which would all share a common LaTeX identifier, and to hide the invocation of the framework machinery. This allowed the use of this high-level analogy through overloading to remain implicit in a casual reading of the text, but of course the underlying framework could be made evident in a more detailed presentation.

6.1.5 Use of elipses

This is a very common idiom in informal mathematics. It normally abbreviates an expression using a recursive or inductive definition. The base case is often left implicit using this sort of idiom, and some results about associativity are often assumed.

For example, a linear sum over a list of arbitrary finite length might be defined

$$\Sigma\,[x_0, x_1, \ldots, x_n] = x_0 + x_1 + \cdots + x_n$$

More formally, we would see this as a recursive/inductive definition.

$$\begin{aligned} \Sigma\,[\,] &= 0 \\ \Sigma\,(x_0 :: X) &= x_0 + (\Sigma\, X) \end{aligned}$$

Still more formally, Σ would not appear in its own definiens, and an elimination operator over lists with special rewrite rules would be used instead. This is the

type-theoretic basis for such definitions, and they are less readable still in this form.

In the case-study, although LEGO does not allow recursive definitions by pattern-matching of the sort seen above, I tended to present definitions in this way since this makes them much more readable. I did this by omitting the code for the definition that actually used the elimination operator, but then demonstrating the interconvertibility of the pairs of terms shown above in the base and inductive cases, thus producing output that had been formally checked by the LEGO system. See page 190 for an example of this.

6.2 Goals in writing, modularity and libraries

In both formal and informal mathematics, the style in which a development is presented depends on the goals that one sets out to reach. Quite often when formalising one may have a single goal in mind: the proof of some particular result. This was certainly true for my case-study, which sought to prove the fundamental theorem of Galois Theory. A similar single-mindedness can often be seen in informal mathematical papers which do the work necessary, and only the work necessary, to reach some interesting result.

Such a development, whether formal or informal, may be read as a narrative account with a definite ending. The way in which the account approaches its conclusion is not necessarily linear; perhaps the underlying subresults can be proved in several different orders. So this narrative method of presentation may also be seen in methods of presentation other than conventional writing, such as hypertext. The key feature however is that everything that is constructed and proved along the way to the final result is designed with that result in mind. The development is a story with an ending.

I note that in a mathematical narrative, unlike a fictional one, it is considered

to be a good idea to give away the ending of the story at the start. Stating the final goals in order to motivate the reader is normal in a mathematical paper and I made an effort to do this in the case-study through use of the specially implemented "we want to prove" feature.

Since the path that such a development follows is made as easy (or as short) as possible, things are likely to be presented in a coherent fashion; however, the development may be so particularly tailored to its one goal that the underlying machinery is not useful for anything else.

In other developments one may not be so interested in proving a particular result as in a general exploration of an area of mathematics and the building of a reuseable library of machinery for it. Many mathematical text books have this sort of aim. Such developments require a slightly different style of presentation; and the way in which material is best presented will depend on which library results are to be proved.

Library use in formalisations has not been very practicable before now, since the proof systems the libraries work in have themselves been subject to constant change and update. However, there are now some powerful and relatively stable systems in existence that will allow the building of large-scale libraries of mathematics[Mizb] to become more of a possibility[QED94].

In order to write a library one requires a good implementation of modularity. In the case of informal mathematics the interface through which the existing resources are used is itself somewhat informal, and this makes such interfacing much easier. However, the misunderstanding and misapplication of a previously proven result is a serious problem of informal mathematics that we might hope to avoid by formalisation. The objective is plausible since there *are* fully formal systems where new projects make use of a library of pre-existing modules; it is common practice in many programming languages, especially ones with an

object-oriented flavour.

In object-oriented programming, one does not tend to mind including the whole of a large library even if one uses only part of it; the flexibility of modules makes up for the waste in including material one does not use. In most proof-checkers, where efficiency is an issue and keeping the size of a development small is more of a concern, this is not so true. Further, extending existing material may necessitate rechecking everything that depends on any of the results already in the library, partly due to the linearity of the proof-checker's context, but also due for the need for library results to share a common context at the time of definition. This can make maintainence a problem. For this and for other reasons that are explained in subsection 6.3.1, my case-study is a stand-alone development that does not use pre-existing libraries.

One interesting related topic I should like to comment on briefly is the relationship between modularity and literateness. When Donald Knuth first developed literate programming, it was used mostly for the documentation of essentially independent projects. Most literate accounts are narratives in the sense described at the start of this section. Now, literate techniques can be applied to libraries also, but a problem occurs that is analagous to the one described above; to get the best designed and most concise account, one wants to document the entire library in an interdependent fashion, but a user may not want to read about functions they do not use, and extending the library may involve some time-consuming rewriting.

The similarity between the problems outlined in the previous two paragraphs is interesting. Since both would occur in the literate formalisation of a library of results, it may warrant further exploration.

6.3 LEGO-specific concerns

Some decisions about the general proof-style that I used were influenced by issues specific to the LEGO proof-checker. Both the particular type theory used in LEGO and the need to keep the development efficient enough to be feasible affected proof style in some ways that are not inherent in the desire to formalise, and I felt the distinction should be made clear.

6.3.1 The LEGO library

The GALOIS project, of which this case-study is a part, aims to develop formalisations in a predicative setting. Although it is easy to use LEGO in a purely predicative way, the type theory contains an impredicative universe of propositions. All the core LEGO libraries use this universe, and for this reason I was not able to use any pre-existing work as a basis for the case-study.

There are other reasons why LEGO is somewhat unsuited to using libraries. Although it has modules, I haven't found them to be very modular in practice. One of the main problems is that of namespace clashes; if the library defines "set" one way, using that identifier in a different way will be denied to anyone trying to build on the library. Building a clean interface between modules is generally harder than one would like. In designing a better system for the development of a lot of mathematical material by many individuals one could probably take some inspiration from object-oriented programming. Other problems with modularity were reviewed in subsection 6.2.

Because of these problems, all machinery in the case-study was built up from scratch. This was not a big task, and gave me absolute control over all details of the development; this was especially helpful when worrying about efficiency issues. However, getting some experience of using the work of others is one of the long-term aims of the GALOIS project, and it is unfortunate that the case-study

has not been able to offer experience with the advantages that would be gained by this (in terms of having ready-made material to hand) and the disadvantages (having to interface with that material.)

6.3.2 Universe ambiguity

A second consequence of predicativity is that one has to pay attention to the level of the type universe in which one it working. In the case-study, a set is defined over an arbitrary element type $el : \mathsf{Type}_0$. This means that the type of subsets of this set must be an element of the next universe up in the hierarchy, Type_1. This means that we cannot form a *set* of subsets directly, since there is a mismatch of levels.

LEGO does allow one to use type universes in an ambiguous fashion. But in its standard mode, it enforces a requirement that all mentions of an ambiguous universe in the use of any particular identifier can be resolved to the same level, causing the sort of restriction explained above. LEGO did have a special mode in which this requirement was not enforced, but unfortunately this slowed the system down by a degree that prevented it being used for developments of more than a very modest size.

In fact, most of the common constructs used in the case-study development involved only the lowest universe of types in the hierarchy, and it wasn't particularly necessary or useful to duplicate these constructs at higher levels.

6.3.3 Inductive types

When implementing mathematical structures as collections of related objects (such as a group, which is implemented as a set together with some operators and proofs that these operators fulfill various axioms), one has a choice of methods.

One could add a layer of abstraction by using inductive types, and writing introduction and elimination operators particular to each structure to relate them to their component objects. Alternatively, one could make use of some form of record structure to bundle the objects together in a more concrete fashion. Record-types are not native to the type theory as I have described it, but records can be implemented through the built-in Σ-type construct, forming the structures by pairing and extracting components from structures by projection.

Ideally, the former approach is probably preferable, since the layer of abstraction allows one to define a clean interface to the structures and to hide their inner implementation in much the same way that one does in object-oriented programming. This would help the modularity of the development. However, an implementation using inductive types turns out to be much slower than the more direct methods using the Σ-type construct, and for efficiency reasons I chose the more direct option.

6.3.4 Freezing

Once a proof-term has been constructed to prove a result, and the reader has observed the structure of the proof, it need not matter what that structure is as far as the rest of the development is concerned. Because of this, we can safely *freeze* the proof; this tells LEGO it should not expand the proof-term in future checking, which can greatly increase the efficiency of the proof-checker when dealing with expressions involving the proof-term.

However, proofs often form only part of an object. For example, in a group, definitions of the group operators are bundled together with proofs that they fulfill the group axioms. The other non-proof information cannot be frozen as it will be required when doing other type-checking. In the group example, when applying the operators from particular groups to arguments we may want to know

their definitions.

For this reason, I break the formation of a composite object such as a particular group into two separate stages; defining the operators, and then proving that they obey the appropriate axioms. This proof would be given a separate name and frozen; then the operators and frozen proof term can be bundled together to define the required group. In this way I ensure that LEGO never wastes time expanding the details of the axiom proofs.

This distinction between proofs and other non-proof objects is usual in informal mathematics, but usually it does not need to be paid explicit attention, as one is not working in a formalism where proofs are themselves objects to be constructed and manipulated.

6.4 The proof style chosen

When comprehending a proof, the requirements of a machine reader and those of a human are likely to differ. For the case-study, the development source must be machine-checked; but I wish to present the checking in such a way that the human reader understands our development easily. I derive a suitable account from a machine-checked source not just through pretty-printing, but also through annotating the source with literate directives that focus on those things that are important for human understanding.

6.4.1 Presenting a proof rather than a result

In [Pol97], Pollack gives an account of what it is necessary to do to persuade a human reader that a certain result has been machine-checked. This has two parts: first, they should be able to check that the proof-term you have constructed does have the type you claim. This should be feasible as long as they can extract

the term and check it in a trusted proof-checker, perhaps one they have written themself. However, they have no need to examine this term; that task may be automated. Secondly, they must be able to read enough of the development that they understand what result has been proved. This is done by expanding the definitions of all the identifiers used in the result either to their most primitive formulation in the type theory or to some familiar and trusted library of basic definitions. Only this much of the development needs to be readable by a human.

In writing mathematics, however, at least some of the interesting material that must be presented is not just *what* has been proved, but also *how* it was proven. The reader wishes to understand the proof, as well as the result. To do this without any danger that a proof had been misread would require the reader to see the definition of every identifier used in the final proof - this would entail reading everything in the development. In essence, the human reader would check the development almost as carefully as would a machine.

However, being sure that every technical detail of the proof checked by the machine is fully understood by the human reader is not a particularly useful aim. When a mathematician reads a paper, it is unlikely that the author needs to communicate every detail of their particular proof for the reading to be a worthwhile experience. What is important is that the reader understands exactly what result has been proved, and has a good enough picture of the structure of the proof that they could fill in the details themself if they so desired. It is only at this level of abstraction that the proof is interesting. Current proof-checkers tend to be capable of smaller steps of reasoning and a lower level of abstraction than their human counterparts.

Therefore, in presenting a formalisation of a proof, the portion of the development that needs to be made human-readable is more than that required to believe the result has been checked, but significantly less than the entire development.

6.4.2 Summarising omitted details

The presentation of the case-study in chapter 10 shows all the important definitions, and the structure of the proofs of the medium-size and major results. But some details of the formalisation which need not be read are replaced with a brief commentary written in a more informal style, or sometimes left implicit altogether.

The account that results is supposed to read somewhat like an informal piece of mathematics supported by precise definitions and statement of results within a formal system, together with good indications of how the proofs were constructed within the formal system without going down to a level of detail that would make the proof hard to read.

In particular, one consequence of this style of presentation is that sometimes typing and conversion judgements about identifiers whose definitions have been omitted should themselves be read as effectively defining the identifiers in question.

This occurs in two main ways. The first is to give the computational content of defined objects whilst suppressing the details of the proof components required to ensure the object is well-formed. For example, suppose one wishes to summarise the definition a subset of some set S. This might be done by introducing the defined object A by giving its type,

▷ A **:** $\mathsf{subset}\ S$

and then demonstrating the predicate that underlies A by saying what it means for a typical element $x : S$ to belong to the subset:

▷ $x \in A \simeq \ldots$

This can be seen as an effective definition of A, with the (normally trivial) proof that set equality is preserved by the subset predicate having been omitted.

A similar presentational style can be used to give the computational content of

a mapping while suppressing the proof that it respects the appropriate equalities:

▷ f : map S S

▷ $f\ x \simeq \ldots$

The second way in which judgements can be used as effective summarised definitions is to succinctly introduce the coercions, components and axioms that comprise some mathematical object. Suppose I have defined a type of partially ordered sets, and wish to introduce it to the reader. I might write:

▷ poset : Type_1

▷ poset $\simeq \langle \Sigma S : \mathsf{set} \rangle\ \langle \Sigma < : \mathsf{rel}\ S \rangle$ poset_axioms

▷ Introduce S : poset

▷ S : set

▷ $\mathsf{refl}_< : \{\forall x, x' \mid S\}\ (x = x') \to x < x'$

▷ $\mathsf{asymm}_< : \{\forall x, y \mid S\}\ (x < y) \to (y < x) \to x = y$

▷ $\mathsf{tran}_< : \{\forall x, y, z \mid S\}\ (x < y) \to (y < z) \to x < z$

Several definitions are implicit in the above summary.

- A coercion to extract the simpler first component (in this case, one of type poset $\to$ set projecting the first component $S_{\cdot\mathbf{1}}$) has been defined.

- A projection for each computational component (in this case, $<$) has also been defined,

▷ $< \simeq [\lambda S \mid \mathsf{poset}]\ S_{\cdot\mathbf{2}\cdot\mathbf{1}}$

- The axioms listed ($\mathsf{refl}_<$, $\mathsf{asymm}_<$, and $\mathsf{tran}_<$) cover all of the properties needed to make up the proof information poset_axioms. Thus we have that

▷ poset_axioms $\simeq \bigwedge_3 (\{\forall x, x' \mid S\}\ (x = x') \to x < x')$
$(\{\forall x, y \mid S\}\ (x < y) \to (y < x) \to x = y)$
$(\{\forall x, y, z \mid S\}\ (x < y) \to (y < z) \to x < z)$

- Finally also the listed axioms have been defined as projections from the proof information component $S_{.\mathbf{2}.\mathbf{2}}$: poset_axioms.

▷ $(\mathsf{refl}_<, \mathsf{asymm}_<, \mathsf{tran}_<) \simeq (S_{.\mathbf{2}.\mathbf{2}}.\mathsf{fst}_3, S_{.\mathbf{2}.\mathbf{2}}.\mathsf{snd}_3, S_{.\mathbf{2}.\mathbf{2}}.\mathsf{thd}_3)$

These sort of conventions having been understood, the summarised definition then contains all the information required in a suitably concise form.

6.4.3 Procedural versus declarative proof style

Sometimes, the tables can turn and give a situation where a human reader requires more details than the machine type-checker in order to follow a proof. This occurs because the relatively smaller size of a human reader's short-term memory can make it hard for them to keep track of the context within which the development works.

This problem is particularly noticeable when trying to follow what John Harrison calls[Har97] a *procedural* style of formal proof as opposed to a *declarative* one. In the procedural style, the proof is constructed by stating a goal and then meeting it by giving a sequence of commands to the checking tool; "Introduce this variable into the context... use this rule on that outstanding goal to convert it into two subgoals..." (This style of proof is called *refinement* proof in LEGO.)

In the declarative style, one builds to ones goal by a process of successively layering incremental definitions and subresults. In many declarative systems, only a statement of the next subresult needs to be given; the proof itself is produced automatically. However one can still see a presentation in which proof-terms are made explicit as being declarative; the feature that distinguishes the style from a procedural one is an absence of proof *commands*.

Refinement proofs are often written interactively in LEGO, with the proof-checker keeping track of the current context and the subgoals left to be proven throughout the interactive session. The extra context provided by having a goal

means that more arguments and terms may be left to be synthesised by the machine rather than the user having to write them explicitly. This produces short proof-scripts; unfortunately they tend to be very hard for a human to read when the proof-checker is not there to remind them what the current goal is at any stage. Whilst declarative proofs are usually more verbose, much of the extra material is actually useful if a reader is to comprehend the structure of the finished proof.

In the case-study, I use a declarative style whenever I wish to expose the reader to the details of the proof. However, often I wish to do more work in one proof step than can be comfortably read once all the details that LEGO requires are supplied. So, I discharge these proof steps by constructing the new incremental subresult by refinement. In this way the reader sees what is being proved (from the type of the new subresult) but does not see the details of this proof.

However, although they are not interested in all the details, the reader may want some clues as to how the proof step is being discharged. I could include the short refinement proof itself, but even small such proofs do not read particularly well. Instead, I list the identifiers that were used in the proof-term that was finally constructed. From this the reader can see which previous results have been used; the way in which they were used can then normally be deduced. This is familiar from informal practice, where one might say "by *this* subresult, we have that *that* holds."

6.4.4 Examples

I have outlined many facets of the style of proof used in the case-study. Concrete examples of all the techniques can be found throughout the rest of the thesis, especially in chapter 10.

Chapter 7

Literate tools

The concept of "a literate formalisation" takes inspiration from the paradigm of literate programming, which was introduced and given that name by Donald Knuth[Knu92]. The philosophy behind literate programming is that although executable by a machine, a computer program should be written and documented in a way that makes it easy for a human reader to understand. This tallies with my stated aims, since although a machine-checked formalisation is normally written as a text source file very much like any other computer program, the development is intended to be read and understood by a human mathematician. The fact that the formalisation has been machine-checked should be evident from reading it only in as much as the reader should know the formal system that they and the machine are working with. Of course this is a lofty ambition, and one which is a long way from being realised.

Presenting a development in a literate manner requires attention at many levels. It needs to be considered with regard to the way the piece of mathematics is formulated, the way in which its concepts are formally implemented and its proofs structured, and the nature of the formal system within which this will be performed. Consideration of these "deeper" matters forms a large part of this thesis. In this chapter, I am more concerned with the translation of the resulting

formalisation from a source in machine-readable format into a form that is tailored for the human reader.

7.1 Implementation framework

In most literate programming environments, the author writes a source which is neither a machine-readable program nor a human-readable document. Instead these items are produced from the source by two translation utilities; the translations are usually known as *tangling* to produce code, and *weaving* to produce the document. In some environments, the tangling translation is avoided by burying all the directives relevant to the weaving process inside the system of comments allowed by the programming language. I chose to use this variation on the approach with my development.

An earlier form of this system that I used was named LaTeXLEGO. This was a simple collection of shell-scripts together with a LaTeX environment for presenting chunks of code. It gave one the ability to write comments that would be processed as chunks of LaTeX documentation when the source modules were woven by the shell-scripts. It was extremely simple, but fairly effective.

One problem with a system like LaTeXLEGO is that although the commentary is supposed to explain the formalisation that it accompanies, there can be no guarantee that the formal development and the informal explanation are related. Since the syntax of the formalisation (that is to say, the ASCII source) is hard to read, a reader usually ends up reading and understanding the author's informal explanation of their formalisation, rather than the formalisation itself. Of course summarising formal results and translating them into less formal language is useful in the sort of communication we are trying to achieve, but if the lexical gap between the formalisation and the commentary grows too large, then it will become hard to relate them to each other.

I sought to address this problem not just by making the formal part of the document easier to read, but also by trying to mix the formalisation and commentary to a greater degree, allowing the use of formal parts within the natural language commentary. Both these solutions require the translator of the chunks of formal source to be able to parse and "understand" it. In order to achieve this, I incorporated the translation process into the proof-checker itself. Thus the altered version of LEGO that I use produces documentation from the source as it is checked.

It is important that the development itself remains checkable by the usual version of the LEGO proof-checker. The source I write could in principle be parsed by standard LEGO so long as the embedded literate directives were removed and one form of definition translated into an equivalent form. But in fact this is not necessary. LEGO itself writes "compiled" versions of the input files it processes in order to speed future processing of those files, and these files are almost free of the directives and definitions to which I refer. There is then a trivial translation from the terms in these files back to terms in the language of standard LEGO by forgetting some embedded information on literate printing.

7.2 Literate techniques

In this section I review the techniques of literate programming that I drew upon for the purposes of making a literate formalisation.

In order to structure a document in a way that made it more pleasing to the reader, Knuth used three main devices. One was a notion of definitional *abbreviation*, achieved through parameterised macros. In the LEGO system, identifier definitions are already used for this purpose, and so I do not need to add anything further to obtain this portion of literateness. The other two devices, which I will consider further here, were those of *re-ordering material* and *pretty-printing*.

7.2.1 Re-ordering material

The purpose of re-ordering material is to get it into the order most convenient for the understanding of the reader, rather than the order required by the machine. In the case of Knuth's literate programming environment for Pascal, this was useful partly in that it allowed parts of the program which were logically related but were necessarily a distance apart in the code because of somewhat arbitrary restrictions on program structures to be presented together. An example would be that it could be necessary to declare a variable a long time before that variable was used. This problem of predefinition does not tend to arise in proof-checkers such as LEGO, which are designed with interactive and open-ended use in mind.

The more general advantage that re-ordering afforded in literate programming is also already present in most proof-checkers, although maybe not to the full extent that one would desire. In presenting a program, reordering is useful in order to show the general structure of the program, breaking it into sub-tasks and indicating their inter-relationship, and then proceeding to work on the sub-tasks. Sometimes it may be useful to present some sub-task independently of the context in which it will be used, and at other times (perhaps more often) one may wish to present the general structure first before describing the details. Re-ordering allows the author to use whichever approach they feel is most appropriate, and to mix and match the "bottom-up" and "top-down" approaches. This interweaving of the two approaches in a development is common in informal mathematical style, and we would hope to use it in a literate formalisation.

Because the meaning of an identifier can change over time depending on the content of the context (due to discharges, other name-scoping, coercions, and so forth), I believe that the order of presentation of the actual formalisation to the machine and to a human reader are best kept fairly similar, to avoid a mismatch in understandings. Therefore I would prefer that reordering be enabled

within the proof-checking system itself rather than through post-processing of the documentation.

Standard LEGO does allow material to be presented in both orders, from the bottom-up through straightforward definition, and from the top-down by construction through refinement. However, some inflexibilities in ordering remain. For example, once one starts to construct a proof or other object, one must proceed to finish the refinement.

Some ways to extend proof-checkers to add to their flexibility are outlined in section 7.3. However, these ideas were largely left unimplemented for this development. I did add to the system the simple ability to formally state and name a goal ahead of time, and to fulfill that goal later on in the development. This style is seen frequently in mathematics, as knowing the goals being sought motivates the reader to follow the development.

In the development, the statements appear in the following way:

▷ We want to prove some_theorem : some_proposition

When the goal is fulfilled later, this appears as

▷ Prove as claimed some_theorem : some_proposition
= the_proof

As far as the main operation of the proof-checker is concerned, this is no different to a normal definition,

▷ Define some_theorem = the_proof : some_proposition

and the only real difference is that the system formally checks that the type of the new definition does match that promised for it earlier on.

A final simple but useful feature of the structuring and ordering part of literate programming is the automatic generation of indexing information. This was just as straightforward to do for a literate formalisation and so I ensured that each identifier definition is automatically indexed by the system.

7.2.2 Pretty-printing

The third and last device is that of *pretty-printing*. Programs (and scripts for machine-checked developments) tend to be sequences of characters from the ASCII alphabet. Programmers know that even in this limited medium, sensible formatting conventions are helpful in making a program comprehensible to a human reader. Once one has greater typographic flexibility to hand, one can make further use of formatting to help the reader. For example, one can reflect the logical role that some identifier has through the font or typeface one uses for it.

Mathematics has made great use of a very large vocabulary of symbols; the increased number of short identifiers available makes expressions lexically small, which helps the reader to parse them. Major use is also made of other formatting conventions, with both dimensions of the page being used, and many sizes of typeface. Sensible conventions for the fixity of operators are also of great assistance.

Although pretty-printing may seem a trivial issue, being to do with mere *style* as opposed to the more important underlying *content*, to this author, its absence in most formal developments seems to be one the main reasons why they are often difficult to read. Most of the ways in which I modified the LEGO proof-checker in order to produce a more literate formalisation could be described as pretty-printing of one form or another.

The reader of this thesis has already seen the most basic features of the pretty-printed output in chapter 3, where the type theory of LEGO was presented, and in other places in the material preceding the current chapter. These features include the use of natural language phrases to introduce expressions and use them in different ways, and the use of different typefaces for identifiers with different roles. Also, an attempt is made to break those expressions that are of many lines length at logical places, based on the depth of nesting of brackets and abstractions

in expressions. (This is imperfect, but the formatting of mathematics is very hard to automate, and this is especially true of expressions that span multiple lines. The line-breaking system implemented works by means of a system of weightings to influence LaTeX's notion of the "badness" of a break, and seems adequate enough.)

Some of the other features are less self-evident, or are deserving of longer discussion. I shall now proceed to explore them. Sometimes these features could have been extended further; I leave a discussion of such extensions until section 7.3.

7.2.2.1 Propositions and types

When using the type theory of LEGO in a predicative fashion, propositions and types share the same universe, and the distinction as to whether some object is a proposition or not is a purely conceptual one, as explained in section 8.1.1. However, I do reflect this distinction through the use of different identifiers for operators, since this is normal outside of type theory.

I also added to the LEGO system a way to mark certain declarations and definitions as propositional, so that the distinction can be made in the literate documentation. For example, if t were some type, I might introduce a term of that type into the context like this:

▷ Introduce x **:** t

But if t were better considered to be a propositional type, I would mark this introduction in my source to indicate that it was the assumption of a hypothesis. Although no different as far as the LEGO proof-checker was concerned, it would then be rendered differently in the documentation:

▷ Suppose x **:** t

A similar distinction works for definitions. In a standard definition, the definiens is important, and the output would have this form:

▷ Define x = long_expression : t

However, when t is a propositional type, this definition would correspond to the construction of a proof object. In this case, the definiens (the proof) may be considered to be less important than the proposition being proved, and I would mark it so that it was rendered differently:

▷ Prove x : t
= long_expression

A conceptual distinction between propositions and other types is also made when choosing abstraction symbols, as explained in the next section.

7.2.2.2 Quantifiers and abstractions

There are three basic forms of abstraction in the type theory of LEGO. These are λ-abstraction, the basic functional abstraction on which the λ-calculus is based; Π-abstraction, the dependent function or product type; and Σ-abstraction, the dependent sum type. In the ASCII-based syntax of LEGO, they are represented by different pairs of brackets that surround the abstracted variable and its type; "`[x:t]`", "`{x:t}`", and "`<x:t>`". In the standard syntax used to represent dependent type theories, they are normally written by prefixing the abstracted variable with the Greek letter after which the abstraction is named; "$\lambda\, x\ :\ t.$", "$\Pi\, x\ :\ t.$", and "$\Sigma\, x\ :\ t.$".

The Greek letters are familiar to us, but the bracketting notation can make expressions easier to read when t is itself a longer term of the type theory and levels of abstraction are nested. I note that another popular notation, that of Martin-Löf, also makes use of brackets in the forming of dependent products. In this notation one would write a Π-abstraction as "$(x\ :\ t)$". However, rounded parentheses are already somewhat overworked as they are also used to override the normal precedence of operators.

I chose to combine the two notations, using both a variety of pairs of brackets

and the Greek letters together in order to denote the abstractions. This notation seems to be easy and natural to read for most people whatever their type-theoretic heritage; I would rate it as a good one.

- ▷ $[\lambda x : t]\ x : t \to t$
- ▷ $\{\Pi x : t\}\ T\ x$: Type
- ▷ $\langle \Sigma x : t \rangle\ T\ x$: Type

When considering a type as a logical proposition, Π-abstraction represents universal quantification, and Σ-abstraction represents existential quantification. I allowed these quantifications to be marked in the development so that they would be rendered using the appropriate symbols in the literate output. As for the other cases of distinction between propositions and types, the different notations reflect a conceptual difference rather than an actual one within the type theory. This is reflected through the brackets used for these abstractions and quantifications sharing the same shape.

Thus if T is considered to be a logical predicate, we might use the alternative quantification symbols:

- ▷ $\{\forall x : t\}\ T\ x \simeq \{\Pi x : t\}\ T\ x$
- ▷ $\langle \exists x : t \rangle\ T\ x \simeq \langle \Sigma x : t \rangle\ T\ x$

7.2.2.3 Identifiers and symbols

In conventional LEGO, the identifiers that can be used for defined and declared objects consist of strings of alphanumeric characters and the underscore character, much as in other programming languages. The pretty-printer does some simple work on these strings by default, setting them in a particular font to indicate that they are local or global, and writing numbers as subscripts.

The vocabulary of mathematical writings is rather larger than this, and it is normal to make use of a great variety of other symbols in order to keep expressions

lexically small. To enable this, I give the pretty-printer a directive to write a LEGO identifier as some particular piece of LaTeX. Thus a term called "`alpha`" in the LEGO source can be denoted as "α" in the pretty-printed output. Although this is a very simple idea, it makes expressions much easier to read.

Being able to write a LEGO identifier as any piece of LaTeX gives the writer a lot of power; power that in principle could be abused. For example, it allows some LaTeX symbol to be overloaded to represent different LEGO terms. Coercions can be used to implement many useful varieties of overloading, but in the absence of a formal system for general overloading, using the same LaTeX symbol to denote two entirely different LEGO identifiers is a quick and simple alternative.

Because it is so powerful, this form of overloading is also completely unsafe. There is no in-built protection against overloaded identifiers introducing ambiguities into the pretty-printed output. For this reason, I use this sort of overloading rather tentatively in the development and am as careful as I can be to make sure that such ambiguities are not introduced. The system also notes in the output when some identifier has been overloaded, so that the reader is aware of the situation. I take some reassurance from the fact that this form of overloading is used widely in informal mathematics, and does not seem to be responsible for many errors in understanding.

7.2.2.4 Fixity and formatting

The mechanism for pretty-printing identifiers explained above works by a simple "search and replace" mechanism; wherever the LEGO identifier occurs in an expression, it is replaced by a piece of LaTeX at that point instead. Therefore this facility does not allow the elements in an expression to be reordered. Some other systems, including Coq, also provide a means for the user to define the grammar that underlies the formal expressions which they write. In principle, this allows

one to write functions of arbitrary fixity.

LEGO is rather less flexible. The original system only allowed a function to be applied postfix using a "dot" application operator. Thus one can write "`x.f`" for "`f x`". This also allows a curried binary function to be written infix, since one can write "`x.plus y`" instead of "`plus x y`". However, although LEGO allows this notation in input, it does not remember it for output. This seemed unfortunate, and so I amended LEGO's output routines so that a function that was originally applied postfix will be applied in that way on output also. Of course I also propogated this change to the pretty-printed output.

I also added some additional directives to affect the fixity and spacing used for function applications in the pretty-printed output. A standard function is applied by writing it before its argument and leaving some spacing between them. A *prefix* function is written immediately preceding its argument, with no space inbetween. A *postfix* function does the same for a function written after its argument. *Infix* functions sit between their two arguments, with extra spacing, whilst *midfix* functions are also written infix but with no additional spacing. This variety of fixities allows most of the pretty-printing effects that I saw in the mathematics that I wanted to formalise.

The following demonstrates the various fixities with which some function, $\Box$, can be applied. By default, the nonfixed application can be written in either of these two ways:

▷ $\Box\ x$ **:** ...

▷ $x.\Box$ **:** ...

Alternatively, $\Box$ can be given particular fixities.

▷ Allow $\Box$ to be written prefix

▷ $\Box x$ **:** ...

▷ Allow $\Box$ to be written postfix

▷ $x\Box$ **:** ...

▷ Allow $\Box$ to be written infix

▷ $x \Box y$ **:** ...

▷ Allow $\Box$ to be written midfix

▷ $x\Box y$ **:** ...

7.2.2.5 Default types

One of the conventions common in mathematical writing is to associate certain identifiers with objects of certain types. Some of these conventions are so common that they may be tacitly assumed in the text. Thus a reader will know that $\sim$ is likely to be a relation, that f is probably a function, and that of two objects called F and V it is likely that F is the field and V the vectorspace, rather than vice versa. Further typing conventions may also be introduced and made use of by an individual text. These conventions enable some information concerning the type of objects to be suppressed in the text.

I sought to allow a similar suppression in the development. Having to write out the type of each object even though it is easy to infer can be particularly distracting within typed abstractions, making expressions longer than one would like. Therefore I allowed any identifier to be associated with a default type. An association is formally noted, and from then on in, when the type of an identifier in an introduction or an abstraction is the default, the type information is suppressed in the pretty-printed output. If the identifier is abstracted over some other non-default type, the full expression is printed as normal.

Thus having declared

▷ Unless otherwise specified, by default S | set

we can introduce the object without mentioning its type:

▷ Introduce S

and we can abstract over it in a similar fashion:

▷ $= : \{\Pi S\}\ S \to S \to \mathsf{prop}$

Additionally, an abstraction in LEGO also contains one other piece of information. The binding may be either explicit or implicit, and this is indicated in the abstraction by the binding symbol used. Because of this, a default includes this binding information as well as a type, and an abstraction using a different binding to the default will be written out in full.

7.2.2.6 Quoting expressions in text

Throughout the development, I make use of sections of text which give an informal overview of the formal development. In order to closely tie them into the formal development, I allow the quoting of formal expressions within the text. Any expression appropriately marked-up in the text is type-checked in the current context by the proof-checker, ensuring that it is well-formed and well-typed, and then it is pretty-printed by the proof-checker just as it would be within a formal statement such as a definition. Declarations can also be introduced locally into the context for use in a particular piece of commentary. Since this also has the added advantage of providing a standard formatting and use of typefaces in formal expressions, I also made use of it elsewhere to put such expressions into other sections of this thesis that have no direct connection to the development itself.

7.3 Further ideas left unimplemented

There are a variety of other ideas and features that could also prove useful in a system for producing literate documentation. Some of these are simple additions such as improving the directives concerning the printing of identifiers in order to handle derived names automatically. For example, currently, having directed

that "`phi`" should be rendered as "ϕ" and have type T by default, the author would have to provide additional directives to handle variants such as "`phi1`" or "`phi'`". Such small changes would be useful and essentially simple to implement but relatively uninteresting. Instead within this section I will review some of the more fundamental advances that could be made.

As mentioned previously, literate programming usually has features in the literate environment to give structure to a document or program. LEGO already has structural features in the form of modules with a notion of dependence, and I did make use of modules to divide my formalisation into sections. However, this module approach is relatively primitive. I have observed elsewhere in this thesis that large-scale formalisations would benefit greatly from more powerful structuring abilities in the proof-checker. If a literate environment is implemented within the proof-checker itself (as I have done for the case-study) in parallel with the new structuring abilities, then this structuring could similarly be shared between the formalisation and the literate documentation.

Part of structuring is the reordering of material. LEGO already allows a term to be introduced as a declaration, so that its content is not determined, and then used in the following material as per normal. When one is ready to give the details of the definition at a later date, the definition can then be *cut* in. However, this process is a very time-consuming one for the proof-checker, involving much rechecking of the subsequent material. I have not found it to be feasible for use in non-trivial developments. A more general ability to leave objects half-constructed, with "holes" remaining in them that are filled at a later date, might be useful.

Another ability related to the re-ordering of material that might be useful would be rapid "context-switching", whereby a group of related local definitions

and declarations could be brought into or out of the context with a single command, allowing multiple definitions in a common context to be made in different parts of the development. Currently all such definitions are best made simultaneously, since the discharge of context items on which earlier terms are dependent means that said context must be reintroduced and the terms then explicitly reapplied to the items when used in later material.

By adding a translation from the formal syntax of terms of LEGO's type theory into a pretty-printed syntax, I have made that syntax much easier to read. However, since the reader of the development sees only the pretty-printed syntax and not the formal syntax beneath it, I believe it is important that the translation can be carried out in the reverse direction also. This translation should itself be formalisable and possible to carry out without relying on intuition or understanding of the material – otherwise one cannot claim that one is really presenting a formalisation. Thus the pretty-printed syntax should be unambiguous. An ad hoc translation of arbitrary LEGO terms into arbitrary LaTeX does not guarantee this.

The main risk of ambiguity comes from unresolvable overloading. In writing the development I was aware of this potential pitfall and so was careful to avoid such ambiguities. Probably I was much more careful than one would usually be when writing mathematics, and the matter does not seem to be a cause for concern there. The fact that the overloading is potentially unsafe could be taken advantage of by a dishonest writer, or could lead to upset if they were foolish, but this remains unimportant so long as this potential is seldom actually realised. Despite finding this reassuring, I do feel the system would be improved further if a general notion of overloading was formalised within the formal implicit syntax of the type theory.

Although the translation from LEGO expressions into pretty-printed LaTeX is

flexible, it is done in a straightforward "search and replace" manner. Because of this, it doesn't allow one to reorder the terms in expressions, or to write operators with arbitrary fixities, or to make as good a use of subscripts and superscripts, as one might like. This could be achieved most easily with the use of a more complicated translation system based on parameterised macros. However, as the pretty-printed syntax diverges further from that of the formal expressions being printed, introducing ambiguities through the translation again begins to be an issue. A better solution might be to allow the definition of arbitrary expression grammars within the LEGO proof-checker itself, as is done in Coq. This would help to ensure that the expressions seen by the reader were merely "prettier" versions of those written by the author and parsed by the proof-checker in order to check the formalisation.

The last idea I want to consider is a further major change in presentation. There are other media that a literate formalisation might be better written in. Since a mathematical development can be read at a variety of levels, and understanding it involves chasing back through layers of definitions and proofs until the reader is satisfied, a hypertext presentation might be a very good medium within which to present such a development. A system could be be directed to produce a web of HTML pages in much the same manner that my own one currently produces a linear LaTeX document.

The only reason I did not take this approach in the development is that using a hyperlinked presentation instead of a linear one involves an additional paradigm shift on top of the one already being attempted. Moving from an informal linear presentation to a formal one (or from an unreadable formal presentation to a literate one, depending on your perspective) already requires that very many new issues be considered. I didn't want to complicate the investigation further with the other fascinating issues involved in moving to a hypertext presentation.

7.4 Source syntax

I present the source for a short sample module so that the reader may see what the files that the author of a development actually writes look like.

```
Module set LaTeX "Sets" Import and rel;


(*@)
A set is a type of elements @[el : Type(0)]@ together with an
equivalence relation defined upon it.
(@*);
[set_axioms [el | Type(0)][eq : rel el] : prop
 = and3 eq.is_refl eq.is_symm eq.is_tran];
[set = <el : Type(0)><eq : rel el> set_axioms eq];
Configure Type S,T, S0,S1,S2,S3,S4 | set;
KindCoercion el = [S : set] S.1;
[equal_in [S : set] = S.2.1 : rel S];
[S | set];
Configure LaTeX eq Infix "$=$";
(*@)
For the sake of readability, I write equality in a set
@[eq | rel S]@ as a standard equals sign, ``@eq@''. The use of
this symbol {\em within} expressions should not be confusing,
as the symbols for definitional equality and convertibility
that are used to relate expressions are written slightly
differently, as ``{\bf =}'' and ``$\cong$'' respectively.
(@*);
[eq = S.2.1 : rel S];
*&[refl = S.2.2.fst3 : eq.is_refl];
```

```
*&[symm = S.2.2.snd3 : eq.is_symm];
*&[tran = S.2.2.thd3 : eq.is_tran];
*&[tran_via [y :: S][x,z || S] = S.2.2.thd3|x|y|z
 : (x.eq y)->(y.eq z)->x.eq z];
Discharge S;
```

Those used to reading LEGO source may have observed some of the small extensions to the standard syntax that I have used in the above example in order to add the literate facility to the proof-checker. I shall briefly review them below.

The "`Module`" command now carries an optional "`LaTeX`" argument that states the title that the section documenting this module should be given. Text to be processed as LaTeX is marked as comments with the special "`(*@)`" and "`(@*)`" delimiters, and contain LEGO expressions marked-up between single "`@`" characters. These are rendered in the current context. Typed terms are introduced temporarily by placing a declaration in this form of expression mark-up; I introduce the identifier `el` and its type explicitly in the first comment, but introduce `eq` silently in the second one by using an implicit binding in order to mention it later. Some identifiers are given the default type `set` with the "`Configure Type`" directive. When `S` is introduced later, its type is suppressed, although it is given in full in the definition of `equal_in` because the abstraction there is an explicit one whereas the default is implicit. The numbers on `S0`, `S1`, ... are pretty-printed as subscripts. The identifier `eq` is made infix and made to be rendered as the symbol "=" in the pretty-printed LaTeX with the "`Configure LaTeX`" directive. The definitions of `refl`, `symm`, and so on are prefixed by an ampersand, "`&`", in order to indicate that they are proof objects. Finally, the abstractions over `x`, `y` and `z` in the definition of `tran_via` are written with the doubled binding symbols "`::`" and "`||`" in order to indicate that they should be pretty-printed with logical universal quantification symbols rather than the usual Π-abstractions.

To conclude, the module is shown in its pretty-printed form.

7.4.0.7 Sets

A set is a type of elements $el : \mathsf{Type}_0$ together with an equivalence relation defined upon it.

▷ Define $\mathsf{set\_axioms} = [\lambda el \,|\, \mathsf{Type}_0]\ [\lambda eq : \mathsf{rel}\ el]$

$\bigwedge_3 eq.\mathsf{is\_refl}\ eq.\mathsf{is\_symm}\ eq.\mathsf{is\_tran} : \{\Pi el \,|\, \mathsf{Type}_0\}\ (\mathsf{rel}\ el) \to \mathsf{prop}$

▷ Define $\mathsf{set} = \langle \Sigma el : \mathsf{Type}_0 \rangle\ \langle \Sigma eq : \mathsf{rel}\ el \rangle\ \mathsf{set\_axioms}\ eq : \mathsf{Type}_1$

▷ Unless otherwise specified, by default $S, T, S_0, S_1, S_2, S_3, S_4 \mid \mathsf{set}$

▷ Define kind-coercion $\mathsf{el} = [\lambda S : \mathsf{set}]\ S_{.\mathbf{1}} : \mathsf{set} \to \mathsf{Type}_0$

▷ Define $\mathsf{equal\_in} = [\lambda S : \mathsf{set}]\ S_{.\mathbf{2}.\mathbf{1}} : \{\Pi S : \mathsf{set}\}\ \mathsf{rel}\ S$

▷ Introduce S

▷ Allow $=$ to be written infix

For the sake of readability, I write equality in a set as a standard equals sign, "=". The use of this symbol *within* expressions should not be confusing, as the symbols for definitional equality and convertibility that are used to relate expressions are written slightly differently, as "$\mathbin{=\!\!=}$" and "$\cong$" respectively.

▷ Define $= = S_{.\mathbf{2}.\mathbf{1}} : \mathsf{rel}\ S$

▷ Prove $\mathsf{refl} : =.\mathsf{is\_refl}$
$= S_{.\mathbf{2}.\mathbf{2}}.\mathsf{fst}_3$

▷ Prove $\mathsf{symm} : =.\mathsf{is\_symm}$
$= S_{.\mathbf{2}.\mathbf{2}}.\mathsf{snd}_3$

▷ Prove $\mathsf{tran} : =.\mathsf{is\_tran}$
$= S_{.\mathbf{2}.\mathbf{2}}.\mathsf{thd}_3$

▷ Prove $\mathsf{tran\_via} : \{\forall y : S\}\ \{\forall x, z \,|\, S\}\ (x = y) \to (y = z) \to x = z$
$= [\lambda y : S]\ [\lambda x, z \,|\, S]\ S_{.\mathbf{2}.\mathbf{2}}.\mathsf{thd}_3 | x | y | z$

▷ Discharge S

Chapter 8

Representation of foundational concepts

This chapter reviews the foundations on which the case-study is built. I explain how I implemented the basic logical and set-theoretic framework within which the case-study proceeds. I also review some of the alternative formulations which I did not choose to use. This chapter also demonstrates some examples of how the coercions, the decisions on proof-style and the pretty-printing techniques explained in chapters 4, 5, 6 and 7 work in practice.

8.1 Logical framework

8.1.1 A predicative basis

In UTT, the type theory underlying LEGO, there is a special impredicative type of propositions, Prop. But for the case-study we have decided to work in a predicative style, and so I do not use this piece of the type theory. This means that propositions and types share the same predicative hierarchy of type universes, namely Type_0, Type_1, and so on. (The motivation for using a predicative and

constructive logical framework was explained in chapter 2.)

There is thus no distinction within the theory itself between proofs and other terms, or between propositions and types. The operators of the type theory work on both sorts of object in the same way, as do all the LEGO proof commands.

However, this identification between proofs and other terms is relatively unfamiliar in informal mathematics, even if there is a precedent in some constructive proof (*e.g.* an existence proof as an algorithm for constructing a witness.) Since I wish for the development to be presented in as familiar a style as possible, I draw a distinction between propositions and types at the level at which they are presented. The hope is that the advantages of the common framework for proofs and non-proof objects can be used without losing the useful conceptual distinction between them.

The distinction is drawn in two ways. As introduced and explained in chapters 3 and 7, different phrasings are used for the introduction and construction of proofs as opposed to non-proofs, and different symbols are used for quantification. Other notational distinctions are implemented by defining two different identifiers for the same term (or, at least, for a pair of intraconvertible terms) and then using the appropriate one of them depending on the circumstances.

As an example of this practice, and as a basis for the case-study framework, I define extra names for the lowest two levels of the type universe hierarchy;

- $\mathsf{prop} \simeq \mathsf{Type}_0$
- $\mathsf{prop}_1 \simeq \mathsf{Type}_1$

Now whenever some type $p : \mathsf{Type}_0$ is meant to be considered as a proposition, it can be type-cast as $p : \mathsf{prop}$ in order that this intention is made clear.

8.1.2 The logical operators

I introduce some proposition and type variables into the context for use in constructing examples in the following presentation.

▷ Unless otherwise specified, by default p, p_1, p_2, p_3 : prop

▷ Unless otherwise specified, by default t, t_1, t_2 : Type_0

▷ Introduce p, p_1, p_2, p_3; t, t_1, t_2 and $\psi : t \to$ prop

Conjunction is implemented as a product; thus the conjunction of two propositions is a non-dependent Σ-type. Logical versions of the pairing constructor and the projection destructors are also defined.

▷ $p_1 \wedge p_2 \simeq p_1 \# p_2$

▷ Suppose $H : p_1 \wedge p_2$

▷ pair H.fst H.snd : $p_1 \wedge p_2$

▷ pair H.fst H.snd $\simeq (H_{\cdot\mathbf{1}}, H_{\cdot\mathbf{2}})$

Larger conjunctions can be built out of the binary one:

▷ $\bigwedge_3 p_1\ p_2\ p_3 \simeq (p_1 \wedge p_2) \wedge p_3$

Implication is written directly as an arrow-type, as is conventional. I also define logical equivalence as the conjunction of two implications.

▷ $p_1 \leftrightarrow p_2 \simeq (p_1 \to p_2) \wedge (p_2 \to p_1)$

Existentional and universal quantifications are written directly as Σ- and Π-types, although the abstractions are tagged so that they use the conventional logical quantification symbols.

▷ $\langle \exists x : t \rangle\ \psi\ x \simeq \langle \Sigma x : t \rangle\ \psi\ x$

▷ $\{\forall x : t\}\ \psi\ x \simeq \{\Pi x : t\}\ \psi\ x$

The types upon which I shall base the remaining logic operators are well-known, but not primitive to LEGO, and so I define them using the inductive types of the theory. I define the disjoint sum of two types, the empty type, and the unit type. They have straightforward introduction and elimination rules

which I shall not detail here.

- ▷ $t_1 \oplus t_2$: Type_0
- ▷ $\bot$: Type_0
- ▷ $\top$: Type_0

When used as propositions, these types represent disjunction, absurdity, and truth, respectively. We can reduce by cases over a disjunction, conclude any proposition from absurdity, and we have a trivial proof of truth. These facts all follow from the more general elimination operators for the primitive types defined above.

- ▷ $\mathsf{inl} : p_1 \to p_1 \vee p_2$
- ▷ $\mathsf{inr} : p_2 \to p_1 \vee p_2$
- ▷ $\mathsf{case} : \{\forall C \mid \mathsf{Type}_1\}\ (p_1 \vee p_2) \to (p_1 \to C) \to (p_2 \to C) \to C$
- ▷ $\mathsf{false} : \mathsf{prop}$
- ▷ $\mathsf{ex\_falso} : \{\forall p\}\ \mathsf{false} \to p$
- ▷ $\mathsf{true} : \mathsf{prop}$
- ▷ $\tau : \mathsf{true}$

Finally, logical negation is contructed in the standard intuitionistic fashion; the negation of a proposition is proved if that proposition implies absurdity.

- ▷ $\neg p \simeq p \to \mathsf{false}$

8.2 Sets

There are other frameworks in which to do mathematics than set theory; for example, it is possible that a category-theoretic framework would be at least as easy to use in a formalisation in type theory. But the Galois theory developed in the case-study is most usually seen in a set-theoretic framework, and so it seemed natural to use such a framework for the development.

Whilst sets are a primitive concept in informal mathematics, they need an explicit implemention in type theory. There are a variety of possibilities. I ended up choosing the most common approach, which is to represent a set as a type with a total equivalence relation defined upon it. Such a structure is often known in the literature of type theory as a *setoid*; the terminology is due to Rod Burstall.

8.2.1 Formalisation

I present this implementation of setoids at a fairly fine level of detail. This provides an example of the general style of construction I use throughout this case-study, which is reviewed in subsection 8.6 at the end of this chapter.

▷ $\mathsf{rel}\ t \simeq t \to t \to \mathsf{prop}$

▷ Introduce $\sim$ **:** $\mathsf{rel}\ t$

▷ $\sim.\mathsf{is\_refl} \simeq \{\forall x : t\}\ x \sim x$

▷ $\sim.\mathsf{is\_symm} \simeq \{\forall x, y \,|\, t\}\ (x \sim y) \to y \sim x$

▷ $\sim.\mathsf{is\_tran} \simeq \{\forall x, y, z \,|\, t\}\ (x \sim y) \to (y \sim z) \to x \sim z$

▷ $\mathsf{set\_axioms} \sim \simeq \bigwedge_3 \sim.\mathsf{is\_refl}\ \sim.\mathsf{is\_symm}\ \sim.\mathsf{is\_tran}$

▷ Define $\mathsf{set} = \langle \Sigma el : \mathsf{Type}_0 \rangle\ \langle \Sigma eq : \mathsf{rel}\ el \rangle\ \mathsf{set\_axioms}\ eq$ **:** Type_1

▷ Discharge $\sim$

▷ Unless otherwise specified, by default $S, T, S_0, S_1, S_2, S_3, S_4$ **:** set

Sets are thus triples of objects: a type, a relation over that type, and a proof of the relation's equivalence properties. I use the identifier set rather than "setoid" since the latter terminology is uncommon outside of type theory, and I wish to hide this implementation layer in the presentation so that wherever possible, the reader can follow the development in a familiar set-theoretic framework.

In order to allow easy quantification over the elements of a set, I introduce a function to coerce a set to its underlying element type.

▷ Define kind-coercion $\mathsf{el} = [\lambda S]\ S.\mathbf{1}$ **:** $\mathsf{set} \to \mathsf{Type}_0$

▷ Introduce S | set

I can now treat S as an abbreviation for its element type; this first (and principal) component of the triple is extracted implicitly by LEGO through the coercive projection el.

The other components are projected by breaking down the Σ-type in a similar manner. However, these projections are not declared as coercions as it is more natural for them to be invoked explicitly. (One *could* write S for equality in S - the system would allow the declaration of a coercion to enable this. But this would be against all mathematical convention.)

▷ = : rel S

▷ refl : =.is_refl

▷ symm : =.is_symm

▷ tran : =.is_tran

▷ Discharge S

Note that the precise structure of the Σ-type used to implement a set (for example, the order of the axioms) is not crucial here. Everything important about the implementation can be observed through the types of the coercions and projections defined upon it (where the projections in question include ones that extract proofs of the axioms satisfied.) In the rest of the case-study presentation, I shall concentrate on these essentials rather than looking at trivial implementation details. Thus I would omit the precise definitions of set, set_axioms and el, and instead note that set : Type_1, and that el is a coercion set → Type_0, and then introduce the four projections above.

Also, coercions are usually left implicit in expressions, and the computations they perform are commonly the natural and obvious ones. They are named only in case the user wishes to refer to them explicitly elsewhere. When this is not the case, it is sometimes more natural to indicate that a coercion has been defined but

to leave it anonymous by, for example, noting that if $S : \mathsf{set}$, then also $S : \mathsf{Type}_0$.

These conventions were also discussed in section 6.4.2.

8.2.2 Alternatives

I considered two alternatives to using total equivalence relations to implement sets. One was to use partial equivalences instead; this allows a cleaner separation of proof information from non-proof information, and affects the way subsets are dealt with. To explain the reasons for avoiding this approach I will first need to show how the concept of subsets is handled within the chosen setoid framework. The other alternative was to use some metatheoretic addition to the framework, such as quotient or congruence types. This introduces an extra layer of abstraction into the reasoning, which can be both a burden (extra work) and a blessing (a clean interface.) For this particular case-study, I thought it best to use the more direct setoid implementation, especially since coercions appear to offer a solution to some of the interface problems it usually entails.

8.3 Mappings

In type theory, a function space is a primitive construct; in fact, it is simply the non-dependent case of the more general Π-type. However, just as sets carry more machinery than types, the type-theoretic implementation of mappings between sets is not just that of functions between their element types. Mappings are functions that are proved to respect the equalities in the sets between which they run. I wish also to define a *set* of mappings between two sets, so I need to say what it means for two mappings to be equal.

8.3.1 Formalisation

▷ Introduce S_1, S_2, S_3

▷ $\mathsf{map}\ S_1\ S_2 : \mathsf{set}$

The elements of the set $\mathsf{map}\ S_1\ S_2$ are functions from S_1 to S_2 that respect the set equalities by sending equal arguments to equal results. The function itself is extracted with an implicit coercion, declared as a Π-coercion so that mappings may be applied to arguments directly. The proof component of a mapping is extracted with an explicit projection, resp.

▷ Introduce $f, g : \mathsf{map}\ S_1\ S_2$

▷ $f : S_1 \to S_2$

▷ $f.\mathsf{resp} : \{\forall x_1, x_2 \,|\, S_1\}\ (x_1 = x_2) \to (f\ x_1) = (f\ x_2)$

Two mappings are equal as elements of the set if they are extensionally equal; that is to say the results returned by applying each of them to any particular argument are equal.

▷ $f = g \simeq \{\forall x : S_1\}\ (f\ x) = (g\ x)$

▷ Discharge g, f

Binary mappings are defined in an analagous way.

▷ $\mathsf{map}_2\ S_1\ S_2\ S_3 : \mathsf{set}$

▷ Introduce $f, g : \mathsf{map}_2\ S_1\ S_2\ S_3$

▷ $f : S_1 \to S_2 \to S_3$

▷ $f.\mathsf{resp}_2 : \{\forall x_1, x_2 \,|\, S_1\}\ (x_1 = x_2) \to \{\forall y_1, y_2 \,|\, S_2\}\ (y_1 = y_2) \to$
$(f\ x_1\ y_1) = (f\ x_2\ y_2)$

▷ $f = g \simeq \{\forall x : S_1\}\ \{\forall y : S_2\}\ (f\ x\ y) = (g\ x\ y)$

I also define specialised versions of the resp_2 projection that focus on only one of the arguments to the binary mapping, leaving the other argument unchanged. These follow from the more general case.

▷ $f.\mathsf{resps}_1 : \{\forall x_1, x_2 \,|\, S_1\}\ (x_1 = x_2) \to \{\forall y : S_2\}\ (f\ x_1\ y) = (f\ x_2\ y)$

▷ $f.\mathsf{resps}_2 : \{\forall x : S_1\}\ \{\forall y_1, y_2 \,|\, S_2\}\ (y_1 = y_2) \to (f\ x\ y_1) = (f\ x\ y_2)$

8.3.2 Alternatives

One obvious generalisation that could have been made would have been to define the concept of a map that took n arguments for any natural number n. (The type of natural numbers is easy to define inductively in LEGO.) Unary and binary mappings, defined individually above, would then be particular cases of the more general definition.

This would have advantages in that all operators, projections and proofs for maps would need to be constructed only once. Since I have defined separate sets of unary maps and binary maps, I have to construct any such objects that they share separately also. In a general library of results, the generalisation from maps of unary and binary arity to maps of arity n would seem an appropriate one to make.

However, this extra layer of abstraction would introduce some inefficiencies and I felt it best to keep everything simple and directly defined in implementing foundational concepts, since they are used so often. For the case-study, the generalisation would not have saved much formalisation work and would have slowed the proof-checking process noticeably.

Another way to implement n-ary mappings in a generic way would be to have them as simple unary maps from a n-fold cartesian product. My main reason for avoiding this implementation idea was that the syntax of LEGO does not accomodate it particularly well. One cannot write an abstraction that gives separate names to the individual components of a tuple. Therefore rather than naming $a : S_1$ and $b : S_2$ by abstracting over a pair as $(a, b) : S_1 \# S_2$, one has to abstract over $x : S_1 \# S_2$ and then use $x.\mathbf{1} : S_1$ and $x.\mathbf{2} : S_2$. Also, using curried mappings rather than unary maps from a product allows a convenient mechanism

for changing fixity: I can use such a mapping f infix by writing $a.f\ b$.

▷ Discharge g, f, S_3, S_2, S_1

8.4 Subsets

The identification of each S : set with an underlying element type means that the typing judgement " $x : S$ " represents the concept that x is a member of the set S. This is convenient in allowing easy abstraction over S through the primitive constructs of the type theory.

However, I would also like to have the concept of set membership be a proposition provable *within* the type theory, since many mathematical results concern questions about whether some element is a member of some set. To do this, we define the type of the subsets of any given set; these are described by membership predicates.

In informal set theory no particular distinction is made between stand-alone sets and sets which are subsets of other sets. I should like to allow a similar blurring in this formalisation, since the idioms it allows are easy to read. I achieve at least a partial success in this aim through the declaration of a few coercions. In particular, the framework coerces any subset to be a set in its own right; this allows membership of the subset to be considered both as a proposition *and* as a typing judgement.

8.4.1 Formalisation

I define the type of subsets of some set, and configure some identifiers to have this as their default type.

▷ Introduce S

▷ subset S : Type_1

▷ Unless otherwise specified, by default A, B, A_1, A_2, A_3 : subset S

A subset is a predicate over the elements of S that satisfies a certain axiom. An element of S that satisfies the predicate is in the subset.

▷ Introduce x, y : S and A

▷ pred A : $S \to$ prop

▷ $x \in A \simeq$ pred A x

The axiom for a subset is that the predicate must respect the equality in the set.

▷ eq_closed$|A$: $\{\forall x_1, x_2 \,|\, S\}$ $(x_1 = x_2) \to (x_1 \in A) \to x_2 \in A$

I consider subsets as sets in their own right using a coercion. The elements are elements of S coupled with a proof of their membership of A. Equality is inherited directly from the set and hence is trivially an equivalence relation.

▷ A : set

▷ A.el $\simeq \langle \Sigma x : S \rangle$ $x \in A$

▷ Introduce a, b : A

▷ $a = b \simeq$ equal_in S $a._{\mathbf{1}}$ $b._{\mathbf{1}}$

There are also two other coercions in this framework that allow me to write expressions involving elements of sets and subsets in a more natural way. Firstly, any element of A can be coerced to be the representative element of S by forgetting the proof that this element is in A.

▷ rep : $A \to S$

Secondly, I define a coercion between proofs that some element of S satisfies the subset predicate, and the element of A that is created by combining the element with this proof. This allows operators defined on A to be indirectly applied to arbitrary elements of S by providing a proof that these elements are in the subset.

▷ make : $(x \in A) \to A$

This pair of coercions rep and make form an example of a general methodology

that is explained in subsection 8.6.

These coercions are used implicitly in expressions throughout the rest of this section. It is hoped that these expressions read naturally as a result.

The evidence that a subset element satisfies the subset predicate is extracted with the ev projection.

▷ $a.\mathsf{ev} : a \in A$

I present the relation of inclusion between subsets of a set, $\subseteq$. Inclusion is easily proved to be reflexive and transitive, and its symmetric closure defines subset equality. I overload the $=$ symbol to represent this equality between subsets, in addition to its current role representing the equality of elements within a set.

▷ Introduce A_1, A_2, A_3

▷ $A_1 \subseteq A_2 \simeq \{\forall x : A_1\}\ x \in A_2$

▷ $\mathsf{subs\_refl} : A \subseteq A$

▷ $\mathsf{subs\_tran} : (A_1 \subseteq A_2) \to (A_2 \subseteq A_3) \to A_1 \subseteq A_3$

▷ $A_1 = A_2 \simeq (A_1 \subseteq A_2) \wedge (A_2 \subseteq A_1)$

I now present a useful type, that of all the subsets included in some specified subset. This allows me to quantify over subsets of A by quantifying over the type $\mathsf{subset\_of}\ A$. A pair of coercions analagous to rep and make are defined.

▷ $\mathsf{subset\_of}\ A : \mathsf{Type}_1$

▷ $\mathsf{subset\_of}\ A \simeq \langle \Sigma B \rangle\ B \subseteq A$

▷ $\mathsf{unsubs} : (\mathsf{subset\_of}\ A) \to \mathsf{subset}\ S$

▷ $\mathsf{make\_subs} : \{\forall B \,|\, \mathsf{subset}\ S\}\ (B \subseteq A) \to \mathsf{subset\_of}\ A$

Many of the concepts that are considered to be defined on sets in an informal presentation are more naturally defined on subsets in this particular formal framework. An example is intersection, $\cap$.

▷ $A_1 \cap A_2 : \mathsf{subset}\ S$

▷ $x \in (A_1 \cap A_2) \simeq (x \in A_1) \wedge (x \in A_2)$

8.4.2 Alternatives

Within informal mathematics, one does not explicitly recognise a distinction between sets and subsets (and between other structures and their substructures.) In the case-study, I kept the concepts distinct within the implementation, but attempted to blur them when useful in practice by coercing subsets into sets. An alternative approach would be to define a type which represented both sets and subsets.

One way to do this would be to include a membership predicate within each set, defined over its underlying element type. This would combine the machinery from the subset and set types together into one type to represent both of them. However, to sensibly compare two such objects or their elements, they would need to share the same underlying type and equality relation. This common "signature" for a pair of sets would play a similar role to that which a set plays for two of its subsets in the case-study framework.

Related to this is the idea of *deliverables*[BMcK93]. This is a general methodology for developments in a type theory that involves a clean separation of computational objects from the proofs of their logical properties. In such a framework, the notion of a setoid is necessarily built around an equality which is a partial equivalence relation rather than a total one. The relation is not reflexive on all elements of the base type. The elements on which equality *is* reflexive are the members of the setoid. Thus the proposition that $x \in A$ is defined to be equivalent to $x = x$.

I used this methodology in my MSc Thesis[Bai94], which was a development of polynomial rings within LEGO. One of the main problems with the approach that I observed at that time was that quantification over the members of a set required both an abstraction over the underlying type, and then a hypothesis that the abstracted element was a member of the set. This verbosity was undesirable.

However, now that coercions are available, this problem could be solved by means similar to those used to make working with subsets more pleasant in the case-study framework.

Nevertheless, adding a propositional component to every set can be awkward in practice since many of the sets one wants to use contain all the elements in their underlying type. In such a case, the predicate is true everywhere, and so each element of the set needs to supply a trivial proof of truth. This redundancy seems somewhat unnatural.

However, the main reason for deciding not to use deliverables in the case-study development is that although the distinction between objects and their properties is a natural one in some cases (such as in proving properties of programs, where this methodology was first developed), and is certainly interesting to apply to conventional mathematics, it is nevertheless not a distinction seen commonly in informal practice. Most interesting mathematical structures are defined as objects that satisfy certain properties. When one defines operators on these structures, one can then assume those properties when one makes the definition. In the deliverables approach, one is forced to make all such operators total in that they must be well-defined on those objects of the correct type that do *not* satisfy the properties in question also. This requirement is not found in the everyday practice of informal mathematics, and so I avoided the deliverables methodology, which would have enforced it.

8.5 Quotients

The type of the arguments to which we apply equality in some S : set itself involves S. (In fact, it is the coercion el applied to S.) Because of this, the set in question can be left implicit and synthesised when the equality is applied. Thus for $x, y : S$ one can write $x = y$ rather than $x.(\mathsf{equal\_in}\ S)\ y$. This idiom

is familiar from set theory, which assumes an intensional "book" equality that is applied to elements of any set. In a setoid framework this equality must be specified explicitly when the set is defined, but having made the definition so the equality can be written in the same style as is used in informal set theory.

The construction of a setoid in type theory by combining an equivalence relation with an existing type is similar to the process of making a quotient of an existing set with a new equivalence relation in set-theoretic mathematics. When a quotient is taken, it is necessary to check that various functions are well-defined on the quotient set; they must respect the new equivalence relation. In implementing mappings between sets, as outlined in subsection 8.3, exactly the same sort of result needs to be proved to show that a function is a mapping.

Because most of the machinery is already present, the implementation of quotient sets in our framework is relatively simple. I first define an equivalence relation over an existing set. This is a symmetric and transitive relation that also relates all elements that are already equal in the original set. A Π-coercion is used to project the relation itself, to allow one to apply the object directly, just as for mappings.

▷ $\mathsf{equiv\_rel}\ S : \mathsf{Type}_1$

▷ Introduce $\sim$: $\mathsf{equiv\_rel}\ S$

▷ $\sim$: $\mathsf{rel}\ S$

▷ $=.\mathsf{is\_subrelation} \sim \simeq \{\forall x, y \mid S\}\ (x = y) \to x \sim y$

▷ $\sim_{.\mathbf{2}} : \bigwedge_3\ (=.\mathsf{is\_subrelation} \sim) \sim.\mathsf{is\_symm} \sim.\mathsf{is\_tran}$

Reflexivity of $\sim$ then follows from the reflexivity of $=$ in S, and so $\sim$ can be used as a new equality on S to form a quotient set.

▷ $S/\sim$: set

▷ $(S/\sim).\mathsf{el} \simeq S.\mathsf{el}$

▷ $\mathsf{equal\_in}\ (S/\sim)\ x\ y \simeq x \sim y$

Any subset A of S can be turned into a subset of $S/\sim$ by considering all the elements that are related by $\sim$ to some $y \in A$. I overload the quotient notation to write this subset as $A/\sim$.

▷ $x \in (A/\sim) \simeq \langle \exists y : A \rangle\ y \sim x$

Note that this construction works even when the original predicate underlying A does not respect the new relation $\sim$. In the case when it does, the predicate underlying $A/\sim$ will be unchanged.

A common construction is to take the quotient of a group by (an equivalence relation induced by) one of its subgroups; this will be defined in subsection 9.3.1.2.

8.6 Construction methodology

Many definitions of new mathematical constructions take the following form: "A *new thing* is an *old thing* such that *this holds*." A familar example from conventional mathematics might be the definition of a commutative group as being a group whose binary operator commutes.

Formally, we might write this definition as follows, using a Σ-type.

▷ Introduce $\mathsf{old\_thing} : \mathsf{Type}$ and $\phi : \mathsf{old\_thing} \to \mathsf{prop}$

▷ Define $\mathsf{new\_thing} = \langle \Sigma X : \mathsf{old\_thing} \rangle\ \phi\ X : \mathsf{Type}$

An example from the formalisation already presented is the element type of a subset A of the set S. An element of A is an element $X : S$ such that $X \in A$. The type is defined as a Σ-type that is in line with the generic structure outlined above.

Mappings and equivalence relations are defined in the same way, although the construction of the particular predicate and Σ-type were not made explicit in the previous presentation. A mapping from S_1 to S_2 is a function $f : S_1 \to S_2$ that respects equality. An equivalence relation on S is a relation $\sim : \mathsf{rel}\ S$ that satisfies various axioms.

The proof information included in new_thing would normally be extracted via some defined projections. The identifiers ev and resp perform this function for subset elements and mappings.

As is indicated by the phrasing of the general definition, having defined the new_thing we may wish to use any X : new_thing as if it were an old_thing. In the example from the first paragraph, a commutative group can be treated as can any other group in an informal development. In the formal framework, I accomodate this by defining a coercion from new_thing back to old_thing. This occurs in all three examples taken so far from the formalisation; subset elements are coerced to set elements, mappings are coerced to functions, and equivalence relations over a set are coerced to plain relations.

I have thus implemented the informal idiom whereby a new_thing is treated as if it were an old_thing by using a coercion. The counterpart to this idiom, whereby an old_thing is treated as a new_thing due to the relevant proof obligations having been discharged, requires more general techniques of overloading. But it can be partially implemented in the existing framework with a second coercion.

▷ Introduce T **:** old_thing $\rightarrow$ Type and f **:** $\{\forall X : \mathsf{new\_thing}\}\ T\ X$

▷ Introduce X **:** old_thing and suppose P **:** $\phi\ X$

Here, f is defined over new_things, and has been given a dependent target type in order to illustrate the most general case. Since we have a proof P that the old_thing called X satisfies the predicate p, there is a new_thing, (X, P), to which we can apply f to obtain a result of type $T\ X$. The idea is that we wish to construct a term of type $T\ X$ by applying f to X, and that we know this application to be well-defined through the existence of the proof term P.

When this idiom is used in informal mathematics, the mechanism works through an apparent overloading of the identifier X. The proof term P needs to be constructed only once, to justify the overloading; from then on a use of X

as if it has type new_thing is enough to reinvoke the previous proof.

The existing system of coercions does not allow this form of proof synthesis directly, although it can offer an explanation and, in simple cases, an implementation for such a mechanism. This works using the idea of overloading an identifier through the use of coercions from a unit type as described on page 108 in chapter 5. We would define the identifier X to have a unit type which could then be overloaded to be either an old_thing or a new new_thing via two coercions. As noted previously though, the coercion system would need to be made more powerful in order to handle this form of overloading in most useful cases.

However, even in the existing framework I can use coercions to get a partial implementation of something similar that allows me to apply f directly to the proof P. This is the technique I make use of in the case-study.

$\triangleright$ Define coercion make_new_from_old $= [\lambda X \,|\, \mathsf{old\_thing}]\ [\lambda P : \phi\ X]$

$(X, P : \mathsf{new\_thing})$ **:** $\{\Pi X \,|\, \mathsf{old\_thing}\}\ (\phi\ X) \to \mathsf{new\_thing}$

Since the type of the application is dependent on the value of the argument of type new_thing, the principal type computed by LEGO will unfortunately involve the proof term P, rather than the original term X to which P will be coerced via (X, P).

$\triangleright$ $f\ P$ **:** $T\ P$

However, in such cases one can use an explicit type-casting to use the more natural type involving X.

$\triangleright$ $f\ P$ **:** $T\ X$

A particular example of this form of construction is given by letting old_thing be the elements of some set S, taking a subset A of S, and setting the predicate over $x : S$ to be that $x \in A$. This makes the element type of the subset A be the new_thing, and the pair of coercions for extracting underlying objects, and for building new things from proofs about the old are rep, and make, respectively.

The construction methodology can be generalised further so that some non-proof information can also be included in the new object. "A *new thing* is an *old thing* called X, with some *machinery on* X, such that *this holds*." For example, a group is a set together with some operators on that set, such that the entire package satisfies certain axioms.

▷ Forget back through ϕ

▷ Introduce $\mathsf{machinery\_on} : \mathsf{old\_thing} \rightarrow \mathsf{Type}$ and ϕ

$: \{\Pi X \,|\, \mathsf{old\_thing}\}\ (\mathsf{machinery\_on}\ X) \rightarrow \mathsf{prop}$

▷ Define $\mathsf{new\_thing} = \langle \Sigma X : \mathsf{old\_thing} \rangle\ \langle \Sigma Y : \mathsf{machinery\_on}\ X \rangle\ \phi\ Y : \mathsf{Type}$

A set forms a good example of this. A set is a type el, together with an equality relation on el, which satisfies the three axioms to make it an equivalence. As for the previous examples, the old_thing (the element type) is extracted with a coercion, and the proof information can be explicitly projected (by the operators refl, symm and tran.) The extra machinery is also extracted with an explicit projection, =. If one ignores the presentational distinction between proof and non-proof objects, both the machinery and the proofs of axioms can all be seen as general extra structure depending on the original old_thing, the element type.

This methodology for constructing new objects by building extra machinery and proof requirements on top of old objects, and defining some explicit projections and also a pair of implicit coercions for assembling and disassembling the new object, will be used often in the case-study development. Another example already presented is that of a subset_of some other subset. In chapter 9, examples will include the algebraic objects such as groups and fields, which are formed by adding new operators and axioms to existing algebraic objects.

Chapter 9

Representation of further mathematical concepts

This chapter looks at the way in which some common mathematical concepts were formalised for the purposes of this development. A few useful results about these are also summarised (but many more were also proved.) The concepts considered include isomorphism and function restriction, decideability and discreteness, the algebraic hierarchy of groups, fields and vectorspaces, finiteness of size and of dimension, and some interrelationships between these notions.

9.0.1 Isomorphism

A first useful notion is that of a set isomorphism (or bijection.) Composition and identity for mappings are first defined in the obvious way:

▷ Introduce S_1, S_2, S_3 : set; f : map S_1 S_2; g : map S_2 S_3 and $x : S_1$

▷ identity$|S_1$: map S_1 S_1

▷ identity $x \simeq x$

▷ $g \circ f$: map S_1 S_3

▷ $g \circ f\ x \simeq g\ (f\ x)$

The isomorphisms are now the subset of functions that have a both-sides inverse.

▷ iso S_1 S_2 : subset (map S_1 S_2)

▷ $f \in$ (iso S_1 S_2)

$\simeq \langle \exists f' : \mathsf{map}\ S_2\ S_1 \rangle\ ((f' \circ f) = \mathsf{identity}) \wedge ((f \circ f') = \mathsf{identity})$

▷ Discharge x, g, f, S_3, S_2, S_1

If f : iso S T, I write f^{-1} for it's inverse in iso T S. I can define a group structure on the subset of isomorphisms iso S S, which will be termed permutations of S, and the notations $\circ$ and $^{-1}$ will be overloaded for use with arbitrary groups.

9.1 Decision

▷ Introduce p : prop

The proposition p is decideable if we have a constructive proof that either it or its negation is true. I write this

▷ p.or_not $\simeq p \vee (\neg p)$

Thus p.or_not can be read as meaning "p is decideable."

9.1.0.1 Definition by cases

If p is a decideable proposition;

▷ Suppose H : p.or_not

then I can work by cases over it.

▷ Introduce S : set and $true\_case, false\_case : S$

▷ if H $true\_case$ $false\_case$: S

▷ IF$_0$: $(\neg p) \rightarrow$ (if H $true\_case$ $false\_case$) $= false\_case$

▷ IF$_1$: $p \rightarrow$ (if H $true\_case$ $false\_case$) $= true\_case$

9.1.1 Decideable subsets

A decideable subset of S is one for which the membership predicate is decideable for all elements of S:

▷ dsubset $S \simeq \langle \Sigma A : \mathsf{subset}\ S \rangle\ \{\forall x : S\}\ (x \in A) \vee (x \notin A)$

▷ Introduce A : dsubset S and $x : S$

▷ $x \in_? A : (x \in A).\mathsf{or\_not}$

Implicit coercions from dsubsets to normal subsets, and from proofs that subsets are decideable to dsubsets, are also defined in the way described in section 8.6.

9.1.1.1 Restriction of individual functions

▷ Introduce f : map S S

I can restrict a function to a decideable subset A of its domain by defining a new function by cases. The restricted function $f \restriction A$ has the same domain set as f but has no effect on elements outside of A.

▷ $f \restriction A$: map S S

▷ $f \restriction A\ x \simeq \mathsf{if}\ (x \in_? A)\ (f\ x)\ x$

9.1.1.2 Restriction of subsets of functions

I can also restrict a whole subset of mappings to A in order to produce a new subset of mappings. I use the same symbol for this as for an individual restriction. (This is an example of a simple overloading of syntax implemented by pretty-printing.)

▷ Introduce F : subset (map S S)

▷ Explicitly overload the identifier $\restriction$

▷ $F \restriction A$: subset (map S S)

▷ $f \in (F \restriction A) \simeq \langle \exists f' : F \rangle\ f = (f' \restriction A)$

▷ Discharge F

Note the nature of this definition: the elements of $F\restriction A$ are equal to existing mappings $f' : F$ that have been restricted to A. This is a useful method of definition that will be used elsewhere also, but it can introduce extra trivial steps of equational reasoning into some proofs. These steps would not normally be noticed in informal mathematical reasoning because of the intensional notion of equality commonly used, but in this formalisation they need explicit attention.

9.1.2 Discreteness

If the equality relation is decideable for all pairs of elements in a set, then this set is defined to be discrete.

▷ $S.\mathsf{is\_discrete} \simeq \{\forall x, y : S\}\ (x = y) \vee (x \neq y)$

One useful result about discreteness is that it is preserved by isomorphism.

▷ DISC_3 **:** $\{\forall S, T : \mathsf{set}\}\ (S.\mathsf{iso}\ T) \to S.\mathsf{is\_discrete} \to T.\mathsf{is\_discrete}$

9.2 Size and finiteness

9.2.1 The natural numbers

9.2.1.1 Definition

The set of natural numbers, $\mathbb{N}$, is defined in a standard way. The elements form an inductive type with constructors 0 and +1, and equality in the set is defined by recursion. This equality can be proved to be decideable and so $\mathbb{N}$ is a discrete set.

▷ 0 **:** $\mathbb{N}$

▷ Introduce m, n **:** $\mathbb{N}$

▷ $n{+}1$ **:** $\mathbb{N}$

▷ $0 = 0 \simeq \mathsf{true}$

- ▷ $0 = n{+}1 \simeq \mathsf{false}$
- ▷ $m{+}1 = 0 \simeq \mathsf{false}$
- ▷ $m{+}1 = n{+}1 \simeq m = n$

Many results can be proved by induction. One useful one is that all predicates respect equality in $\mathbb{N}$.

- ▷ $\mathsf{nat\_eq\_resp} : (m = n) \to \{\forall p : \mathbb{N} \to \mathsf{prop}\}\ (p\ m) \to p\ n$

Thus every predicate on $\mathbb{N}$ forms a subset.

9.2.1.2 Ordering

Two orders are defined on $\mathbb{N}$. The "less than" ordering $<$ is defined by recursion,

- ▷ $0 < 0 \simeq \mathsf{false}$
- ▷ $0 < n{+}1 \simeq \mathsf{true}$
- ▷ $m{+}1 < 0 \simeq \mathsf{false}$
- ▷ $m{+}1 < n{+}1 \simeq m < n$

$<$ is easily proved to be irreflexive and transitive. The reflexive and antisymmetric weaker order $\leq$ is defined in terms of $<$:

- ▷ $m \leq n \simeq (m < n) \vee (m = n)$

9.2.1.3 Canonical subsets

The reason for defining these relations is to allow me to construct canonical subsets with n elements. These will be used to define a notion of finite size.

- ▷ $\mathsf{canonical\_subset}\ n : \mathsf{subset}\ \mathbb{N}$
- ▷ $m \in (\mathsf{canonical\_subset}\ n) \simeq m < n$

By defining the canonical_subset construction as a coercion, I am able to write various nice abbreviations by treating any natural number n as a subset, and hence (by the as_a_set coercion) also as a set of n distinct indices.

9.2.1.4 Tuples

I can now define a set of n-tuples over S very simply as the maps from n (as a set of indices) to S. For $i : n$, projecting the ith member from the tuple $\boldsymbol{x}$ is achieved simply by applying the map $\boldsymbol{x}$ to i.

▷ $S^{\wedge}n \simeq \mathsf{map}\ n\ S$

▷ Introduce $\boldsymbol{x} : S^{\wedge}n$ and $i : n$

▷ $\boldsymbol{x}_{\backprime}i \simeq \mathsf{ap}\ \boldsymbol{x}\ i$

Another way to define the set of n-tuples over S would have been as the n-fold cartesian product of S with itself. However, in such a framework, the action of projection must be defined by induction over the subset of the natural numbers less than n. This is awkward to formulate and to reason about, and I much prefer this direct implementation of tuples as mappings from indexing sets.

I define analogues of zero and successor within the canonical indexing sets.

▷ $\underline{0}\ n : n{+}1$

▷ $i{+}\underline{1} : n{+}1$

I also define operations to take the head and tail of non-empty tuples.

▷ Introduce $\boldsymbol{y} : S^{\wedge}n{+}1$

▷ $\mathsf{hd} : \mathsf{map}\ (S^{\wedge}n{+}1)\ S$

▷ $\mathsf{tl} : \mathsf{map}\ (S^{\wedge}n{+}1)\ (S^{\wedge}n)$

▷ $\mathsf{hd}\ \boldsymbol{y} \simeq \boldsymbol{y}_{\backprime}\underline{0}$

▷ $(\mathsf{tl}\ \boldsymbol{y})_{\backprime}i \simeq \boldsymbol{y}_{\backprime}i{+}\underline{1}$

Tupling operations are easily defined to lift functions to work on tuples (by composition of the function and the tuple to make a new tuple), and to lift subsets of a set to become subsets of tuples. I overload the symbol $^{\wedge}$

▷ $f^{\wedge}n : \mathsf{map}\ (S^{\wedge}n)\ (S^{\wedge}n)$

▷ $(f^{\wedge}n\ \boldsymbol{x})_{\backprime}i \simeq f\ (\boldsymbol{x}_{\backprime}i)$

▷ $A^{\wedge}n : \mathsf{subset}\ (S^{\wedge}n)$

▷ $\boldsymbol{x} \in (A^{\wedge}n) \simeq \{\forall i : n\}\ (\boldsymbol{x}_{\diagdown} i) \in A$

▷ Discharge $\boldsymbol{y}, i, \boldsymbol{x}$

9.2.2 Function tools

I define also a set of further tools to use on sets (some discrete), subsets (some decideable), functions and tuples, such as excluding an element from a subset, adding an element into a subset, isomorphisms between subsets to which this has been done, isomorphisms to swap a pair of elements in a set, and Kronigger δ tuples. These tools will not be seen in the presentation. I mention them so that you know what mechanisms below the surface are actually used to prove the expected properties about finite size and dimension. Details are found in the appendix.

9.2.3 Size

My notion of size is defined on subsets (it turns out that we do not need to consider the sizes of objects in the development, only the sizes of subobjects of these objects.) Two subsets (possibly of different sets) are said to have the same size if there is an isomorphism between them.

▷ Introduce T : set and B : subset T

▷ $A \cong B$: prop

▷ $A \cong B \simeq \mathsf{iso}\ B\ A$

I write this propositional relation with a new symbol $\cong$ in order to indicate that it is the existence of the isomorphism, rather than it's computational content, that is the matter of interest here. $\cong$ is easily proved to be reflexive, symmetric and transitive.

9.2.3.1 Finiteness

A subset is finite if it has the same size as one of the canonical n-element subsets.

▷ $A.\mathsf{is\_finite} \simeq \langle \exists n : \mathbb{N} \rangle\ A \cong n$

9.2.3.2 Comparing sizes

I define one subset to be smaller than another if they are both finite and one has a smaller size than the other.

▷ $A \preccurlyeq B \simeq \langle \exists a, b : \mathbb{N} \rangle \bigwedge_3 (A \cong a)\ (B \cong b)\ (a \leq b)$

9.2.3.3 Results about finite size

Many standard results about finiteness and size can be proved, although the proofs are quite complicated and make copious use of the set of tools described in subsection 9.2.2 above. I prove first that finite size is unique;

▷ EQSIZE_1 **:** $(m \cong n) \to m = n$

The $\preccurlyeq$ relation is clearly transitive. From EQSIZE_1 it also follows that it is antisymmetric.

▷ $\mathsf{smaller\_asymm}$ **:** $(B \preccurlyeq A) \to (A \preccurlyeq B) \to A \cong B$

Two results relating decideability and finiteness, and $\cong$ and subset equality, are also proved.

▷ FIN_1 **:** $B.\mathsf{is\_finite} \to (A.\mathsf{decideable\_subs}\ B) \to A \preccurlyeq B$

▷ FIN_3 **:** $(A \subseteq B) \to A.\mathsf{is\_finite} \to (A \cong B) \to A = B$

It also follows from DISC_3 that all finite sets are discrete.

▷ DISC_2 **:** $A.\mathsf{is\_finite} \to A.\mathsf{is\_discrete}$

9.3 Algebras

So far I have considered only sets and subsets. I now build some more structure on top of sets to turn them into groups, abelian groups, and fields. More levels in this hierarchy of objects would allow some concepts to be defined at a more general level, but would also require extra processing time. For the purposes of the case-study, I therefore kept the hierarchy as simple as possible.

9.3.1 Groups

9.3.1.1 Definition

A group is a set with an identity element id, an associative composition operator $\circ$, and an inversion operator $^{-1}$.

▷ group

$\simeq \langle \Sigma G : \mathsf{set} \rangle\ \langle \Sigma id : G \rangle\ \langle \Sigma \circ : \mathsf{map}_2\ G\ G\ G \rangle\ \langle \Sigma^{-1} : \mathsf{map}\ G\ G \rangle\ \mathsf{group\_axioms}$

▷ Explicitly overload the identifier $\circ$

▷ Allow $\circ$ to be written infix

▷ Explicitly overload the identifier $^{-1}$

▷ Allow $^{-1}$ to be written postfix

The axioms are as follows:

▷ Introduce G **:** group

▷ $\mathsf{o\_assoc}$ **:** $\{\forall g, h, j : G\}\ ((g \circ h) \circ j) = (g \circ (h \circ j))$

▷ $\mathsf{o\_id}$ **:** $\{\forall g : G\}\ (g \circ \mathsf{id}) = g$

▷ $\mathsf{id\_o}$ **:** $\{\forall g : G\}\ (\mathsf{id} \circ g) = g$

▷ $\mathsf{o\_inv}$ **:** $\{\forall g : G\}\ (g \circ g^{-1}) = \mathsf{id}$

Left inversion can be proved from right inversion:

▷ $\mathsf{inv\_o}$ **:** $\{\forall g : G\}\ (g^{-1} \circ g) = \mathsf{id}$

A subgroup is a subset of the group that is closed with respect to the three

operators. I define decideable subgroups and subgroups of other subgroups just as I did for subsets.

▷ subgroup $G \simeq \langle \Sigma U : \mathsf{subset}\ G \rangle$

$\bigwedge_3 (\mathsf{id} \in U)\ (\{\forall g, h : U\}\ (g \circ h) \in U)\ (\{\forall g : U\}\ g^{-1} \in U)$

▷ dsubgroup $G \simeq \langle \Sigma U : \mathsf{subgroup}\ G \rangle\ U.\mathsf{is\_decideable}$

▷ Introduce U **:** subgroup G and g, h **:** G

▷ subgroup_of $U \simeq \langle \Sigma D : \mathsf{subgroup}\ G \rangle\ D \subseteq U$

9.3.1.2 Quotient by a subgroup

A common way in which to quotient a group G is by the equivalence relation of being in the same coset of some subgroup U.

▷ $\approx$-U **:** equiv_rel G

▷ $g.(\approx\text{-}U)\ h \simeq (g^{-1} \circ h) \in U$

I use a coercion in order to write the subgroup itself in order to represent this equivalence relation. This allows me to use the standard notation for this form of taking a quotient:

▷ G/U **:** set

9.3.1.3 Normality of subgroups

I define the standard operation of conjugation within a group.

▷ $g \divideontimes h \simeq h^{-1} \circ (g \circ h)$

This allows me to define the condition of one subgroup being normal in another subgroup.

▷ Introduce D **:** subgroup G

▷ $D.\mathsf{normal\_in}\ U \simeq \{\forall h : U\}\ \{\forall g : D\}\ (g \divideontimes h) \in D$

▷ Discharge D, h, g, U, G

9.3.1.4 Abelian groups

An abelian group is one in which the composition operator commutes.

▷ $\mathsf{abelian\_group} \simeq \langle \exists G : \mathsf{group} \rangle \ \{\forall g, h : G\} \ (g \circ h) = (h \circ g)$

I use additive notation for the operators in abelian groups in order to emphasise this commutativity.

▷ Introduce G **:** $\mathsf{abelian\_group}$ and g, h **:** G

▷ $\mathbf{0} \simeq \mathsf{id}$

▷ $g + h \simeq g \circ h$

▷ $-g \simeq g^{-1}$

I also define the minus operator as the addition of a negated element:

▷ Explicitly overload the identifier $-$

▷ $g - h \simeq g + (-h)$

9.3.2 Fields

9.3.2.1 Definition

A field is an abelian group with a commutative and associative product $\times$ that distributes through $+$, together with an identity (unit) element $\mathbf{1}$ and a reciprocation operator $^1\!/$ that acts as the inverse of $\times$ on all non-zero elements.

▷ $\mathsf{field} \simeq \langle \Sigma F : \mathsf{abelian\_group} \rangle \ \langle \Sigma \mathbf{1} : F \rangle \ \langle \Sigma \times : \mathsf{map}_2 \ F \ F \ F \rangle \ \langle \Sigma {}^1\!/ : \mathsf{map} \ F \ F \rangle$
$\mathsf{field\_axioms}$

I define the field to be non-trivial, and I also define the reciprocation operator to leave the zero element $\mathbf{0}$ unchanged, rather than only defining it on non-zero elements. This axiomatisation will make the field equality decideable, but since all fields considered in the development will be discrete, this is acceptable. Having the reciprocation operator defined on the whole field makes equational reasoning using it much easier.

▷ Introduce F **:** field

▷ times_assoc **:** $\{\forall x, y, z : F\}\ ((x \times y) \times z) = (x \times (y \times z))$

▷ times_comm **:** $\{\forall x, y : F\}\ (x \times y) = (y \times x)$

▷ times_plus **:** $\{\forall x, y, z : F\}\ (x \times (y + z)) = ((x \times y) + (x \times z))$

▷ times_un **:** $\{\forall x : F\}\ (x \times \mathbf{1}) = x$

▷ nontriv **:** $\mathbf{1} \neq \mathbf{0}$

▷ times_recip **:** $\{\forall x : F\}\ ((x = \mathbf{0}) \wedge ((^1/x) = \mathbf{0})) \vee ((x \times (^1/x)) = \mathbf{1})$

The standard collection of subobjects are defined.

▷ subfield $F \simeq \langle \Sigma K : \text{subgroup } F \rangle$

$\bigwedge_3 (\mathbf{1} \in K)\ (\{\forall x, y : K\}\ (x \times y) \in K)\ (\{\forall x : K\}\ (^1/x) \in K)$

▷ dsubfield $F \simeq \langle \Sigma K : \text{subfield } F \rangle\ K$.is_decideable

▷ Introduce K **:** subfield F

▷ subfield_of $K \simeq \langle \Sigma J : \text{subfield } F \rangle\ J \subseteq K$

9.3.2.2 Vectorspaces

A vectorspace over F is an abelian group of vectors that can be acted on by the field of scalars with the operator $*$.

▷ F-space $\simeq \langle \Sigma V : \text{abelian\_group} \rangle\ \langle \Sigma * : \text{map}_2\ F\ V\ V \rangle$ space_axioms

▷ Introduce V **:** F-space

▷ sc_plus **:** $\{\forall x : F\}\ \{\forall v, w : V\}\ (x * (v + w)) = ((x * v) + (x * w))$

▷ plus_sc **:** $\{\forall x, y : F\}\ \{\forall v : V\}\ ((x + y) * v) = ((x * v) + (y * v))$

▷ times_sc **:** $\{\forall x, y : F\}\ \{\forall v : V\}\ ((x \times y) * v) = (x * (y * v))$

▷ un_sc **:** $\{\forall v : V\}\ (\mathbf{1} * v) = v$

Any field may be considered as a vectorspace over itself by taking products in the field to be the scaling operator. This construction is defined as a coercion so it can be used transparently.

▷ F **:** F-space

▷ $*|F \simeq \times$

9.4 Span and dimension

The Galois case-study will involve vectorspaces of finite dimension. I thus need to formalise what this means.

9.4.1 Linear sums

Suppose I have some G : abelian_group. I can define a linear sum across n-tuples over G by induction.

▷ Introduce $\boldsymbol{x}$ **:** $G\hat{}n+1$

▷ $\sum () \simeq \mathbf{0}$

▷ $\sum \boldsymbol{x} \simeq (\mathsf{hd}\ \boldsymbol{x}) + (\sum (\mathsf{tl}\ \boldsymbol{x}))$

▷ Discharge $\boldsymbol{x}$

9.4.1.1 Linear combinations

We can form the linear combination of a tuple of scalars with a tuple of vectors by lifting the scaling operator $*$ to work on tuples, and then forming the linear sum of the result.

▷ Introduce $\boldsymbol{x}$ **:** $F\hat{}n$; $\boldsymbol{v}$ **:** $V\hat{}n$ and w **:** V

▷ $\boldsymbol{*} \simeq *\hat{}n$

▷ $\sum (\boldsymbol{x} \boldsymbol{*} \boldsymbol{v})$ **:** V

9.4.1.2 Spans

I now consider the span of the tuple of vectors $\boldsymbol{v}$ under some subset K of scalars.

▷ $\boldsymbol{v}$-span_over K **:** subset V

▷ $w \in (\boldsymbol{v}\text{-span\_over } K) \simeq \langle \exists \boldsymbol{x} : K\hat{}n \rangle\ (\sum (\boldsymbol{x} \boldsymbol{*} \boldsymbol{v})) = w$

9.4.2 Finite-dimensional spaces

9.4.2.1 Independence

A tuple of vectors $\boldsymbol{v}$ is said to be independent over some subset of scalars K if no vector in it can be written as a linear combination of the others. This is phrased as a fact about zero sum linear combinations:

▷ independent_over K **:** subset $(V\hat{}n)$

▷ $\boldsymbol{v} \in$ (independent_over K)

$\simeq \{\forall \boldsymbol{x} : K\hat{}n\}\ ((\sum\ (\boldsymbol{x} * \boldsymbol{v})) = \mathbf{0}) \to \{\forall i : n\}\ (\boldsymbol{x}_{\backslash} i) = \mathbf{0}$

9.4.2.2 Bases

▷ Introduce W **:** subset V

$\boldsymbol{v}$ is a basis over K for W if it is an independent tuple of vectors in W that spans W over K.

▷ W.has_basis_over K $\boldsymbol{v}$

$\simeq \bigwedge_3\ (\boldsymbol{v} \in (W\hat{}n))\ (W = (\boldsymbol{v}\text{-span\_over } K))\ (\boldsymbol{v} \in (\text{independent\_over } K))$

In such a case we say that W has finite dimension n over K:

▷ W.has_fin_dim_over K $n \simeq \langle \exists \boldsymbol{v} : V\hat{}n \rangle$ W.has_basis_over K $\boldsymbol{v}$

or simply that the vectorspace W is finite-dimensional over K:

▷ W.is_fin_dim_over $K \simeq \langle \exists n : \mathbb{N} \rangle$ W.has_fin_dim_over K n

It can be proved that the dimension of a finite-dimensional vectorspace is unique. In other words, all bases for W over K have the same size. Thus we may talk about *the* finite dimension of W over K.

▷ Discharge

$W, w, \boldsymbol{v}, \boldsymbol{x}, V, K, F, h, g, G, B, n, m, f, x, A, false\_case, true\_case, S, H, p$

Chapter 10

The case-study development

In this chapter I present an account of the case-study development. The actual mathematics is not particularly difficult, and should be understandable without a detailed knowledge of the formal framework. The minimal prequisites for understanding the presentation should thus be a grounding in simple algebra and an understanding of the basic syntax of UTT expressions and judgements as presented in chapter 3. Special attention should be given to the way in which the conversion judgement "$\simeq$" is used to summarise definitions; this is explained in subsection 6.4.2. Reading chapter 2, which gives an overview of the general approach taken to proving the result in question is also advised since the Galois theory is developed in a constructive setting.

To understand the more formal part of this presentation more fully, it is helpful to have read more of this thesis. Chapter 4 must be browsed to understand how coercions are used. Chapter 6 on proof style, and chapters 8 and 9 on the implementation of the mathematical framework within which the development takes place also give a useful background.

Although the presentation is not at quite as high a level as that usually found in a mathematics paper, it is still made at a relatively high-level compared to the underlying formalisation, with many details suppressed. The full details of the

formalisation are given in the long appendix to this thesis. It may be useful to refer to this appendix to chase identifiers back to their definitions, but the hope is that this will not often be necessary to understand and believe[Pol97] the case-study, only to clarify any ambiguities or to examine details of the formalisation for the sake of interest.

The proofs themselves are presented at a relatively high-level too; the direction and structure of the larger proofs are described by the kind of natural language explanations common in an informal mathematical paper. However these explanations do involve formal statements that have been checked by the machine. To illustrate that the material may be presented at other levels also, an auxiliary proof is presented at a level between that of this chapter and that of the uncommented appendix in the following chapter 11.

10.1 Definitions and setting

A standard presentation of Galois theory works with individual fields, later extending and comparing them so that the originals become subfields of each other and of these extensions. In this presentation I found it useful instead to start by introducing a universal field which contains all the other fields I will consider. These can then be worked on directly as decideable subfields of the universal field.

▷ Introduce F | field

Although I work in a constructive setting, all fields are taken to be discrete.

▷ FIELD_1 : $\{\forall x, y : F\}\ (x = y) \vee (x \neq y)$

10.1.1 Subfield morphisms

10.1.1.1 Definition of the automorphisms of a subfield

This section defines the group of automorphisms that act on a subfield of F as a subgroup of its permutation group.

▷ Introduce L : subfield F

10.1.1.1.1 Previously defined material

I can define the group perm of permutations of a set and the subgroup of Perm of permutations that act only on a subset of that set.

▷ perm F : group

▷ Perm L : subgroup (perm F)

▷ Introduce g : perm F and x : F

▷ g.fixes $x \simeq (g\ x) = x$

▷ $g \in (\mathsf{Perm}\ L) \simeq \{\forall x : F\}\ (x \in L) \vee (g.\mathsf{fixes}\ x)$

When L is a decideable subfield of F, any element of perm L (*i.e.* a permutation of L where L is considered as a set in its own right) can be extended to be an element of Perm L defined on the whole of the universal field F, by setting it to be the identity outside of L. I can also prove that

▷ PERMc : $\{\forall f : \mathsf{Perm}\ L\}\ \{\forall x : L\}\ (f\ x) \in L$

and so Perm L and perm L can be considered to be essentially equivalent within this setting. Thus for decideable subfields L, in this development I shall use the associated subgroups Perm L, since these are all included in a universal group perm F in the same way that all the subfields L are included in the universal field F.

Another building block will be the subset of mappings that distribute through a binary operation $\oplus$ on some subset.

▷ Introduce $\oplus$: $\mathsf{map}_2\ F\ F\ F$ and f : map $F\ F$

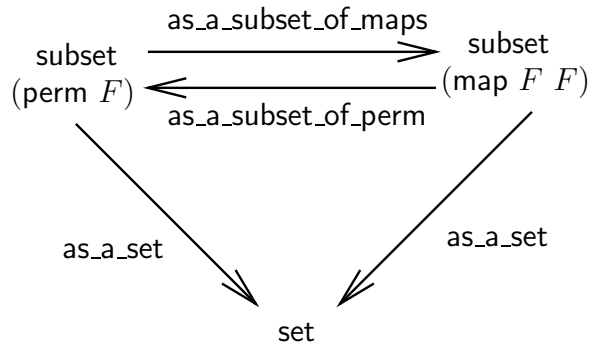

Figure 10.1: An incoherency problem

▷ $\mathsf{resp\_map}_2 \oplus L$: subset (map F F)

▷ $f \in (\mathsf{resp\_map}_2 \oplus L) \simeq \{\forall x, y : L\}\ (f\ (x \oplus y)) = ((f\ x) \oplus (f\ y))$

Now since the equality on the set of permutations of F is essentially the same as that on the set of ordinary mappings, any predicate defining a subset of mappings (such as that above) can in turn define a subset of permutations. A similar construction works in the reverse direction.

▷ Introduce G : subset (perm F) and A : subset (map F F)

▷ $g \in A.\mathsf{as\_a\_subset\_of\_perm} \simeq g.\mathsf{rep} \in A$

▷ $f \in G.\mathsf{as\_a\_subset\_of\_maps} \simeq \langle \exists g : G \rangle\ f = g.\mathsf{rep}$

▷ Discharge $A, G, f, \oplus, x$

Having to make the moves between subsets of map F F and subgroups of perm F explicit in this presentation is unfortunate. (So also is having to fulfill the trivial but tedious proof obligations that such moves induce, though the details are suppressed in this chapter.) I would consider this to be the greatest problem with the framework I chose for the case-study development. However, I did not find another framework without worse problems, or a way to define the constructions as_a_subset_of_perm and as_a_subset_of_maps as coercions without introducing incoherencies. These arise because the equivalent subsets of perm F and map F F will be coerced to distinct sets, as shown in figure 10.1.

10.1.1.1.2 The permutations that are automorphisms of L

An automorphism of L is a permutation of the subfield that respects the operations $+$ and $\times$ on L. I define the subset of maps that respect these operations on L:

▷ Define resp_plus_times $=$ (resp_map$_2$ $+$ L) $\cap$ (resp_map$_2$ $\times$ L)

: subset (map F F)

The subset of subfield automorphisms can now be defined as the intersection of the permutations on L with this subset. I introduce a temporary coercion to allow the direct application of members of this subset.

▷ Define Aut_subset $=$ (Perm L) $\cap$ resp_plus_times.as_a_subset_of_perm

: subset (perm F)

▷ Let coercion Aut_rep $= [\lambda g :$ Aut_subset$]$ g.rep : Aut_subset $\rightarrow$ perm F

Perm L already forms a subgroup. Now, since permutations of a subset can be proved to be closed on that subset by the result PERMc, I can prove that membership of resp_plus_times also is closed for subset epimorphisms under the group operators id, $\circ$ and $^{-1}$. It then follows that Aut_subset forms a subgroup.

▷ Aut : subgroup (perm F)

▷ $g \in$ Aut $\simeq$ $g \in$ Aut_subset

Two canonical coercions are also defined that allow one to consider subfield automorphisms as subset permutations and as subfield homomorphisms.

▷ Aut_Perm : Aut $\rightarrow$ Perm L

▷ Aut_hom : Aut $\rightarrow$ subfield_hom L

▷ Discharge g, L

10.1.1.2 Subfield morphisms that fix a second subfield

Given K, J : subfield F, I define the subset of maps which are subfield morphisms that act only on J, and that also fix everything in K.

▷ Allow -fix_hom to be written postfix

▷ Define -fix_hom

$= [\lambda K, J : \mathsf{subfield}\ F]\ (\mathsf{map\_from}\ J) \cap ((\mathsf{resp\_plus\_times}\ J) \cap (\mathsf{fixing}\ K))$

$: (\mathsf{subfield}\ F) \to (\mathsf{subfield}\ F) \to \mathsf{subset}\ (\mathsf{map}\ F\ F)$

▷ Introduce K, J : subfield F and g : K-fix_hom J

These are the basic axioms that g satisfies:

▷ $g.\mathsf{ev.fst} : \{\forall x : F\}\ (x \in J) \vee (g.\mathsf{fixes}\ x)$

▷ $g.\mathsf{ev.snd.fst.fst} : \{\forall x, y : J\}\ (g\ (x + y)) = ((g\ x) + (g\ y))$

▷ $g.\mathsf{ev.snd.fst.snd} : \{\forall x, y : J\}\ (g\ (x \times y)) = ((g\ x) \times (g\ y))$

▷ $g.\mathsf{ev.snd.snd} : \{\forall x : K\}\ g.\mathsf{fixes}\ x$

Recall that any field can be considered as a vectorspace over itself. In fact, the field will form an algebra, since multiplication is defined on both the scalars and the vectors. When $K \subseteq J$, the subset K-fix_hom J contains maps which are morphisms on J as an algebra over K, since they leave scalars in K unaffected, and distribute through the scaling multiplication. Thus for $x : K$, $v : J$ one has that $(g\ (x * v)) = (x * (g\ v))$ and so g is an algebra homomorphism. This can be a useful way in which to view the role of the subset K-fix_hom J.

▷ Discharge g, J, K

10.1.1.3 Definition of fixing subgroups and fixed subfields

▷ Introduce L : subfield F

The Fundamental Theorem of Galois Theory will state that a certain Galois connection between subsets of F and subsets of perm F is an anti-isomorphism for some classes of these objects (certain special subfields and subgroups.) I now define this connection.

The connection could be made at several levels. The level at which I work is relative to a subfield L. The connection then runs between subfields contained in L and subgroups contained in Aut L.

▷ Introduce G : subgroup_of (Aut L)

Elements $x : F$ fixed by all automorphisms in G form a subfield. (It is closed under the field operators because all $g : G$ respect these operators, and so fixedness under such g will be preserved by them.) Now take the part of this subfield within that is within L; this defines the fix operator that forms one half of the connection.

▷ fix L G : subfield_of L

▷ $x \in (\text{fix } L\ G) \simeq (\{\forall g : G\}\ g.\text{fixes } x) \wedge (x \in L)$

▷ Introduce K : subfield_of L and g : perm F

Conversely, morphisms g that fix everything in K form a subgroup. (The proof that the composition and inverses of morphisms will fix all $x : L$ that the original morphisms do is straightforward.) The aut operator returns the part of this subgroup that is within Aut L, and defines the second half of the connection.

▷ aut L K : subgroup_of (Aut L)

▷ $g \in (\text{aut } L\ K) \simeq (\{\forall x : K\}\ g.\text{fixes } x) \wedge (g \in (\text{Aut } L))$

When the subfield L is clear from the context, I will denote fix L G and aut L K by the abbreviations $G^{\triangledown}$ and $K^{\triangle}$, respectively.

▷ Discharge g, K, x, G, L

10.1.1.4 The Galois connection between fixing subgroups and fixed subfields

▷ Introduce L | subfield F

I now need to prove that $^{\triangledown}$ and $^{\triangle}$ form a Galois connection between subgroups of Aut L and subfields of L. The proofs are straightforward; for example it, is clear that if $U \subseteq G$ then any $x : F$ fixed by everything in G is fixed by everything in U, and so on. The original inclusions extend to show that equal subgroups fix equal subfields and that equal subfields have equal fixing subgroups.

▷ Construct by refinement GC_1
: $\{\forall U, G \mid \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)\}\ (U \subseteq G) \to G^{\triangledown} \subseteq U^{\triangledown}$
($^{\triangledown}$, ev, $\in$, fixes, snd, fst, pair, perm, $\subseteq$, Aut, subgroup_of)

▷ Construct by refinement $\mathsf{GC}_1\mathsf{eq}$
: $\{\forall U, G \mid \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)\}\ (U = G) \to U^{\triangledown} = G^{\triangledown}$
(perm, $\subseteq$, fst, GC_1, snd, $^{\triangledown}$, pair, $=$, Aut, subgroup_of)

▷ Construct by refinement GC_2 : $\{\forall J, K \mid \mathsf{subfield\_of}\ L\}\ (J \subseteq K) \to K^{\triangle} \subseteq J^{\triangle}$
($^{\triangle}$, perm, ev, Aut, $\in$, fixing_group, snd, fst, pair, $\subseteq$, subfield_of)

▷ Construct by refinement $\mathsf{GC}_2\mathsf{eq}$
: $\{\forall J, K \mid \mathsf{subfield\_of}\ L\}\ (J = K) \to J^{\triangle} = K^{\triangle}$
($\subseteq$, fst, GC_2, snd, $^{\triangle}$, perm, pair, $=$, subfield_of)

▷ Construct by refinement GC_3 : $\{\forall G : \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)\}\ G \subseteq G^{\triangledown\triangle}$
($^{\triangledown}$, ev, $\in$, fixes, fst, Aut, perm, fixing_group, pair, subgroup_of)

▷ Construct by refinement GC_4 : $\{\forall K : \mathsf{subfield\_of}\ L\}\ K \subseteq K^{\triangle\triangledown}$
($^{\triangle}$, perm, ev, Aut, $\in$, fixing_group, fst, fixes, pair, subfield_of)

Every Galois connection can also be proved to satisfy the following two equalities by combining these previous results together.

▷ Prove GC_5 : $\{\forall G : \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)\}\ G^{\triangledown} = G^{\triangledown\triangle\triangledown}$
$= [\lambda G : \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)]\ \mathsf{pair}\ (\mathsf{GC}_4\ G^{\triangledown})\ (\mathsf{GC}_1\ (\mathsf{GC}_3\ G))$

▷ Prove GC_6 : $\{\forall K : \mathsf{subfield\_of}\ L\}\ K^{\triangle} = K^{\triangle\triangledown\triangle}$
$= [\lambda K : \mathsf{subfield\_of}\ L]\ \mathsf{pair}\ (\mathsf{GC}_3\ K^{\triangle})\ (\mathsf{GC}_2\ (\mathsf{GC}_4\ K))$

▷ Discharge L

10.1.2 Galois groups and subfields

10.1.2.1 Definition of a Galois group

A Galois group on L is a finite subgroup of $\mathsf{Aut}\ L$ with a fixed subfield over which L is finite-dimensional.

▷ Introduce L : subfield F

▷ Define galois_group $= \langle \Sigma G : \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L) \rangle\ G.\mathsf{is\_galois\_group} : \mathsf{Type}_1$

▷ Introduce G : galois_group L

The axioms that define G as a Galois group are

▷ gg_finite G : G.is_finite

▷ gg_findim G : L.is_fin_dim_over G^{∇}

There is the usual coercion from proofs to objects:

▷ make_gg : $\{\Pi G \mid \text{subgroup\_of } (\text{Aut } L)\}$ G.is_galois_group $\rightarrow$ galois_group L

▷ Discharge G, L

10.1.2.2 Definition of a Galois subfield

▷ Introduce L : subfield F

▷ Define galois_subfield $= \langle \Sigma K : \text{subfield } F \rangle$ K.is_galois_subfield L : Type_1

A Galois subfield of L is equal to the fixed subfield of some Galois group.

▷ Introduce K : galois_subfield L and let $G = K$.gf_gg : galois_group L

▷ gf_equal : $G^{\nabla} = K$

Again, there is a coercion from proofs to objects.

▷ make_gf : $\{\Pi K \mid \text{subfield } F\}$ $(K$.is_galois_subfield $L) \rightarrow$ galois_subfield L

▷ Discharge G, K, L

10.1.3 Some special subfields

10.1.3.1 Introduction of a particular subfield

The first two-thirds of the Fundamental Theorem will concern a decideable subfield of our universal field F. I introduce it into the context now.

▷ Introduce L | dsubfield F

10.1.3.2 A coercion to aid readability

I introduce a useful coercion that will help me to write expressions used in the following work in a natural fashion. It uses the transitivity of subset inclusion to

turn any subgroup of G : galois_group L into a subgroup of Aut L.

▷ subg_trans : $\{\Pi G \mid \text{galois\_group } L\}$ (subgroup_of G) $\rightarrow$ subgroup_of (Aut L)

10.1.3.3 Definition of a K-subfield

For $K \subseteq L$, a K-subfield is a decideable subfield of L that contains K, and also has some properties when the subfields are considered as vectorspaces:

▷ Define -subfield $= [\lambda K : \text{subfield\_of } L]\ \langle \Sigma J : \text{dsubfield\_of } L \rangle\ J.\text{is\_-subfield } K$
: (subfield_of L) $\rightarrow$ Type_1

▷ Introduce K : subfield_of L and J : K-subfield

The four axioms that define J as a K-subfield are:

▷ sub_subs J : $J \subseteq L$

▷ subs_sub J : $K \subseteq J$

▷ sub_fin_dim_over K J : J.is_fin_dim_over K

▷ fin_dim_over_sub J : L.is_fin_dim_over J

If L.is_fin_dim_over K, then classically any $J \subseteq L$ containing K would have these properties. But in a constructive formalisation of the Fundamental Theorem, I need to state them explicitly, as they do not hold in general and are necessary in my proofs.

There is also the usual coercion from proofs to objects.

▷ make_-subfield : $\{\Pi J \mid \text{dsubfield\_of } L\}$ (J.is_-subfield K) $\rightarrow$ K-subfield

▷ Discharge J, K

10.2 Statement of the Fundamental Theorem

I can now state the main result that I will prove, the Fundamental Theorem of Galois Theory. It falls into three thirds, each of which I divide into a further three parts.

The first two thirds show that $^{\nabla}$ and $^{\Delta}$ form an anti-isomorphism between the Galois groups and subfields of L.

▷ Introduce G : galois_group L

First comes a subresult regarding certain subgroups of G,

▷ Let $\mathsf{ft}_1\mathsf{i} = \{\forall U : \mathsf{subgroup\_of}\ G\}\ (U.\mathsf{is\_decideable\_in}\ G) \rightarrow$

$(L.\mathsf{is\_fin\_dim\_over}\ U^{\nabla}) \rightarrow U.\mathsf{is\_galois\_group} : \mathsf{prop}_1$

Then a pair of results shows the first half of the anti-isomorphism:

▷ Let $\mathsf{ft}_1\mathsf{ii} = G^{\nabla}.\mathsf{is\_galois\_subfield}\ L : \mathsf{prop}_1$

▷ Let $\mathsf{ft}_1\mathsf{iii} = G = G^{\nabla\Delta} : \mathsf{prop}$

▷ We want to prove $\mathsf{FT}_1 : \bigwedge_3\ \mathsf{ft}_1\mathsf{i}\ \mathsf{ft}_1\mathsf{ii}\ \mathsf{ft}_1\mathsf{iii}$

▷ Introduce K : galois_subfield L

The second third is complementary to the first, showing a subresult concerning K-subfields, and completing the anti-isomorphism.

▷ Let $\mathsf{ft}_2\mathsf{i} = \{\forall J : K\text{-subfield}\}\ J.\mathsf{is\_galois\_subfield}\ L : \mathsf{prop}_1$

▷ Let $\mathsf{ft}_2\mathsf{ii} = K^{\Delta}.\mathsf{is\_galois\_group} : \mathsf{prop}$

▷ Let $\mathsf{ft}_2\mathsf{iii} = K = K^{\Delta\nabla} : \mathsf{prop}$

▷ We want to prove $\mathsf{FT}_2 : \bigwedge_3\ \mathsf{ft}_2\mathsf{i}\ \mathsf{ft}_2\mathsf{ii}\ \mathsf{ft}_2\mathsf{iii}$

The final third concerns K-subfields of L. There is a logical equivalence between the properties of K being a Galois subfield with respect to such a subfield J and the normality of J's fixing subgroup J^{Δ} in K^{Δ}.

▷ Introduce J : K-subfield

▷ Let $\mathsf{ft}_3\mathsf{A} = K.\mathsf{is\_galois\_subfield}\ J : \mathsf{prop}_1$

▷ Let $\mathsf{ft}_3\mathsf{B} = J^{\Delta}.\mathsf{normal\_in}\ K^{\Delta} : \mathsf{prop}$

Also in this circumstance, restriction to J forms an isomorphism of groups between the quotient group K^{Δ}/J^{Δ} and $\mathsf{aut}\ J\ K$.

▷ We want to prove FT_3 **:** $(\mathsf{ft}_3\mathsf{A} \leftrightarrow \mathsf{ft}_3\mathsf{B}) \wedge$

$(\mathsf{ft}_3\mathsf{A} \rightarrow \langle \exists \pi : \mathsf{iso}\ (K^{\triangle}/J^{\triangle})\ (\mathsf{aut}\ J\ K)\rangle\ (\{\forall g : K^{\triangle}/J^{\triangle}\}\ (g\restriction J) = (\pi\ g)) \wedge$

$(\{\forall g, h : K^{\triangle}/J^{\triangle}\}\ ((h \circ g)\restriction J) = ((h\restriction J) \circ (g\restriction J))))$

▷ Discharge $\mathsf{ft}_3\mathsf{B}, \mathsf{ft}_3\mathsf{A}, J, \mathsf{ft}_2\mathsf{iii}, \mathsf{ft}_2\mathsf{ii}, \mathsf{ft}_2\mathsf{i}, K, \mathsf{ft}_1\mathsf{iii}, \mathsf{ft}_1\mathsf{ii}, \mathsf{ft}_1\mathsf{i}, G$

10.3 Proof of the Fundamental Theorem

10.3.1 Some subresults

10.3.1.1 Lemma 1

I state and prove a simple lemma relating restriction to the taking of a quotient with respect to a fixing subgroup.

10.3.1.1.1 Statement of Lemma 1

I divide the lemma into two parts.

▷ Introduce G **:** $\mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)$; J **:** $\mathsf{dsubfield\_of}\ L$ and let $U = G \cap J^{\triangle}$

: $\mathsf{subgroup}\ (\mathsf{perm}\ F)$

▷ We want to prove $\mathsf{LEMMA}_1\mathsf{i}$

: $\{\forall g_1, g_2 : G\}\ (g_1.(\approx\text{-}U)\ g_2) \leftrightarrow ((g_1\restriction J) = (g_2\restriction J))$

▷ We want to prove $\mathsf{LEMMA}_1\mathsf{ii}$ **:** $(G\restriction J) \cong (G/U)$

10.3.1.1.2 Proof of the first part

The proof proceeds via the logical equivalence of a sequence of propositions.

▷ Introduce g_1, g_2 **:** G

By definition,

▷ $g_1.(\approx\text{-}U)\ g_2 \simeq ({g_1}^{-1} \circ g_2) \in U$

Since $({g_1}^{-1} \circ g_2) \in G$,

▷ $\mathsf{Q}_1 : ((g_1{}^{-1} \circ g_2) \in U) \leftrightarrow ((g_1{}^{-1} \circ g_2) \in J^{\triangle})$

▷ $\mathsf{Q}_2 : ((g_1{}^{-1} \circ g_2) \in J^{\triangle}) \leftrightarrow (\{\forall x : J\}\ (g_1{}^{-1} \circ g_2\ x) = x)$

I also have that

▷ $\mathsf{Q}_3 : (\{\forall x : J\}\ (g_1{}^{-1} \circ g_2\ x) = x) \leftrightarrow (\{\forall x : J\}\ (g_1\ x) = (g_2\ x))$

▷ $\{\forall x : J\}\ (g_1\ x) = (g_2\ x) \simeq g_1.(=\restriction J)\ g_2$

▷ $\mathsf{Q}_4 : (g_1.(=\restriction J)\ g_2) \leftrightarrow ((g_1 \restriction J) = (g_2 \restriction J))$

Thus the logical equivalence is proved.

▷ Discharge to prove as claimed $\mathsf{LEMMA}_1\mathsf{i}$
$: \{\forall g_1, g_2 : G\}\ (g_1.(\approx\text{-}U)\ g_2) \leftrightarrow ((g_1 \restriction J) = (g_2 \restriction J))$
using
$(\mathsf{Q}_1.\mathsf{iff\_tran}\ \mathsf{Q}_2).\mathsf{iff\_tran}\ (\mathsf{Q}_3.\mathsf{iff\_tran}\ \mathsf{Q}_4)$
$: ((g_1{}^{-1} \circ g_2) \in U) \leftrightarrow ((g_1 \restriction J) = (g_2 \restriction J))$

▷ Discharge g_2, g_1

10.3.1.1.3 Proof of the second part

$\mathsf{LEMMA}_1\mathsf{i}$ allows me to define an isomorphism,

▷ $\theta : \mathsf{iso}\ (G/U)\ (G \restriction J)$

The definition is as follows. In the forward direction: each element of the quotient G/U has a representative in G,

▷ Introduce $g : G/U$ and let $g' = (g.\mathsf{ev})_{.\mathbf{1}} : G$

The result of applying θ to g is the restriction of its representative g' to J:

▷ $\theta\ g \simeq g' \restriction J$

Since there is an isomorphism between the two subsets, they are by definition of equal size.

▷ Prove as claimed $\mathsf{LEMMA}_1\mathsf{ii} : (G \restriction J) \cong (G/U)$
$= \theta$

▷ Discharge U, J, G

10.3.1.2 Theorem A

▷ Introduce G **:** galois_group L and let $K = G^{\nabla}$ **:** subfield_of L

This theorem separates into three parts. The proofs of the first two parts have not been formalised in the case-study due to a lack of time. The reader and the proof-checker should assume this pair of results without proof for this case-study.

10.3.1.2.1 Statement of Theorem A

The first part relates the number of distinct algebra morphisms over some $J : K$-subfield to the dimension of J over K, and can be proved using Dedekind's Lemma.

▷ Assume without proof $\mathsf{THM\_A}_1$

: $\{\forall J : K\text{-subfield}\}\ (K\text{-fix\_hom}\ J) \preccurlyeq (\text{sub\_fin\_dim\_over}\ K\ J).\text{basis}$

The second part follows from a result about the free generation of any vectorspace over L on which a finite automorphism subgroup such as G acts linearly from the fixed part under G of this vectorspace considered as a space over K.

▷ Assume without proof $\mathsf{THM\_A}_2$

: $\{\forall U : \text{subgroup\_of}\ G\}\ \langle \Sigma H : U^{\nabla}.\text{is\_fin\_dim\_over}\ K\rangle\ (G/U) \cong (\text{basis}\ H)$

The third part, which I shall prove, is a corollary to the second.

▷ We want to prove $\mathsf{THM\_A}_3$

: $\langle \exists n : \mathbb{N}\rangle\ \langle \exists \boldsymbol{w} : F^{\wedge} n\rangle\ (L.\text{has\_basis\_over}\ K\ \boldsymbol{w}) \wedge (G \cong \boldsymbol{w})$

10.3.1.2.2 Proof of the corollary

Consider the trivial subgroup containing only the identity permutation on F.

▷ $\{\mathsf{id}\}$ **:** subgroup_of G

▷ $g \in \{\mathsf{id}\} \simeq \mathsf{id} = g$

The corollary will follow by applying $\mathsf{THM\_A}_2$ to this subgroup.

▷ Prove subresult P_0 **:** $\langle \Sigma H : \{\mathsf{id}\}^{\nabla}.\text{is\_fin\_dim\_over}\ K\rangle\ (G/\{\mathsf{id}\}) \cong (\text{basis}\ H)$
$= \mathsf{THM\_A}_2\ \{\mathsf{id}\}$

Since

▷ $\mathsf{P}_1 : \{\mathsf{id}\}^{\triangledown} = L$

we have that

▷ $\mathsf{P}_2 : L.\mathsf{has\_basis\_over}\ K\ (\mathsf{basis}\ H)$

Now by definition,

▷ $g_1.(\approx\text{-}\{\mathsf{id}\})\ g_2 \simeq ({g_1}^{-1} \circ g_2) \in \{\mathsf{id}\}$

and so it follows from the uniqueness of inverses that any pair of elements of $\mathsf{perm}\ F$ related by $\approx$-$\{\mathsf{id}\}$ are equal.

▷ $\mathsf{P}_3 : \mathsf{is\_same\_relation}\ (\approx\text{-}\{\mathsf{id}\})\ (\mathsf{equal\_in}\ (\mathsf{perm}\ F))$

Thus by means of a canonical isomorphism between G and $G/=$ we can conclude that

▷ $\mathsf{P}_4 : G \cong (G/\{\mathsf{id}\})$

Therefore the basis $\boldsymbol{w} = (\mathsf{basis}\ H)$ satisfies the properties we require.

▷ Prove as claimed $\mathsf{THM\_A}_3$
$: \langle \exists n : \mathbb{N} \rangle\ \langle \exists \boldsymbol{w} : F^{\wedge} n \rangle\ (L.\mathsf{has\_basis\_over}\ K\ \boldsymbol{w}) \wedge (G \cong \boldsymbol{w})$
$= \mathsf{THM\_A}_3\mathsf{\_proof}$

▷ Discharge but keep K, G

10.3.1.3 Theorem B

▷ Introduce $J : K\text{-}\mathsf{subfield}$

Note that since J is a K-subfield there exists $\boldsymbol{w}$ such that

▷ $\mathsf{W} : J.\mathsf{has\_basis\_over}\ K\ \boldsymbol{w}$

10.3.1.3.1 Statement of Theorem B

Theorem B divides into five related parts. Firstly there is a prelude to establish that $G{\restriction}J$ is a finite subset.

▷ We want to prove $\mathsf{THM\_B}_0 : G{\restriction}J.\mathsf{is\_finite}$

Secondly, the main part of the theorem proves three results as a consequence of the facts that a number of other objects also have the same finite size.

▷ We want to prove $\mathsf{THM\_B}_1$ **:** $G{\restriction}J = (K\text{-}\mathsf{fix\_hom}\ J)$

▷ We want to prove $\mathsf{THM\_B}_2$ **:** $G{\restriction}J \cong \boldsymbol{w}$

▷ We want to prove $\mathsf{THM\_B}_3$ **:** $J = J^{\triangle\triangledown}$

Finally I will prove a corollary to B1 by setting J to be L.

▷ We want to prove $\mathsf{THM\_B}_4$ **:** $(G{\restriction}L) = (K\text{-}\mathsf{fix\_hom}\ L)$

10.3.1.3.2 Preliminaries

I define U : $\mathsf{subgroup\_of}\ G$ as the part that fixes J:

▷ $U \simeq G \cap J^{\triangle}$

Applying $\mathsf{THM\_A}_2$ to this subgroup gives me a basis over K for $U^{\triangledown}$.

▷ $\mathsf{THM\_A}_2\ U$ **:** $\langle \Sigma H : U^{\triangledown}.\mathsf{is\_fin\_dim\_over}\ G^{\triangledown}\rangle\ (G/U) \cong (\mathsf{basis}\ H)$

▷ Let $\boldsymbol{v} = \mathsf{basis\ H}$ **:** $U^{\triangledown\wedge}(\mathsf{dim\ H})$

10.3.1.3.3 A cycle of inequalities

I now prove a cycle of inequalities concerning the sizes of G/U, $G{\restriction}J$, K-$\mathsf{fix\_hom}\ J$, $\boldsymbol{w}$ and $\boldsymbol{v}$.

By LEMMA_1ii I have

▷ a **:** $G/U \cong G{\restriction}J$

I can prove that $G{\restriction}J$ is a decideable subset of K-$\mathsf{fix\_hom}\ J$. (The decideability proof is based on a result SPAN_4 the proof of which I shall use as an example in the next chapter.)

▷ b_1 **:** $G{\restriction}J.\mathsf{decideable\_subs}\ K\text{-}\mathsf{fix\_hom}\ J$

It then follows from FIN_1 that $G{\restriction}J$ is smaller, and also that it is itself finite.

▷ b **:** $G{\restriction}J \preccurlyeq K\text{-}\mathsf{fix\_hom}\ J$

▷ Prove as claimed $\mathsf{THM\_B}_0$ **:** $G{\restriction}J.\mathsf{is\_finite}$
$= \mathsf{SMALLER}_0\ \mathsf{b}$

$\mathsf{THM\_A}_1$ applied to J provides the next inequality:

▷ $\mathsf{c} : K\text{-}\mathsf{fix\_hom}\ J \preccurlyeq \boldsymbol{w}$

Now I prove that the span of $\boldsymbol{v}$ over K contains that of $\boldsymbol{w}$. Firstly,

▷ $\mathsf{d}_1 : (\boldsymbol{w}\text{-}\mathsf{span\_over}\ K) \subseteq J$

because $K \subseteq J$ and each element of $\boldsymbol{w}$ is in J. By GC_4,

▷ $\mathsf{d}_2 : J \subseteq J^{\triangle\triangledown}$

and since $U \subseteq J^{\triangle}$, by GC_1 in turn

▷ $\mathsf{d}_3 : J^{\triangle\triangledown} \subseteq U^{\triangledown}$

Lastly $U^{\triangledown} \subseteq (\boldsymbol{v}\text{-}\mathsf{span\_over}\ K)$ by definition. Thus I have

▷ $\mathsf{d}_{123} : (\boldsymbol{w}\text{-}\mathsf{span\_over}\ K) \subseteq (\boldsymbol{v}\text{-}\mathsf{span\_over}\ K)$

and since $\boldsymbol{v}$ and $\boldsymbol{w}$ are K-bases and hence independent, I can conclude that

▷ $\mathsf{d} : \boldsymbol{w} \preccurlyeq \boldsymbol{v}$

Finally, $\mathsf{THM\_A}_2$ proved that

▷ $\mathsf{e} : \boldsymbol{v} \cong G/U$

By the antisymmetry of $\preccurlyeq$ this chain of inequalities allows me to conclude all these objects have the same size:

▷ $\mathsf{b'} : G{\restriction}J \cong K\text{-}\mathsf{fix\_hom}\ J$

▷ $\mathsf{c'} : K\text{-}\mathsf{fix\_hom}\ J \cong \boldsymbol{w}$

▷ $\mathsf{d'} : \boldsymbol{w} \cong \boldsymbol{v}$

10.3.1.3.4 Conclusion

The required results now follow. If a subset of a finite set has the same size as the whole set, then it is the whole set.

▷ Prove as claimed $\mathsf{THM\_B}_1 : G{\restriction}J = (K\text{-}\mathsf{fix\_hom}\ J)$
$= \mathsf{FIN}_3\ \mathsf{b}_1\mathsf{.snd}\ \mathsf{THM\_B}_0\ \mathsf{b'}$

The second of the three main results is immediate:

▷ Prove as claimed $\mathsf{THM\_B}_2 : G{\restriction}J \cong \boldsymbol{w}$
$= \mathsf{b'.eqsize\_tran}\ \mathsf{c'}$

For the third, I have that $J \subseteq J^{\Delta\nabla}$ and need to prove the reverse inclusion. This follows since by d_{123} the span of $\boldsymbol{w}$ is contained in that of $\boldsymbol{v}$, but by d' I know that $\boldsymbol{v}$ and $\boldsymbol{w}$ have the same size. It follows that the spanned subsets U^{∇} and J are equal; then as $J^{\Delta\nabla} \subseteq U^{\nabla}$ the inclusion is proved.

▷ Prove as claimed $\mathsf{THM\_B_3} : J = J^{\Delta\nabla}$
$= \mathsf{THM\_B_3\_proof}$

▷ Discharge J

For the corollary to $\mathsf{THM\_B_1}$, I set J to be equal to the whole of L considered as a K-subfield.

▷ $\mathsf{J} \simeq L$

▷ Prove as claimed $\mathsf{THM\_B_4} : (G{\restriction}L) = (K\text{-}\mathsf{fix\_hom}\ L)$
$= \mathsf{THM\_B_1}\ \mathsf{J}$

▷ Discharge K, G

10.3.2 The main proof

10.3.2.1 Proof of first part of the fundamental theorem

▷ Introduce G : galois_group L and U | subgroup_of G

▷ Suppose H_1 : U.is_decideable_in G and H_2 : L.is_fin_dim_over U^{∇}

10.3.2.1.1 Proof of $\mathbf{FT_1i}$

The first result follows straight from the hypotheses. U is finite since it is a decideable subset of finite G,

▷ Prove subresult $\mathsf{P_1} : U.\mathsf{is\_finite}$
$= \mathsf{SMALLER_0}\ (\mathsf{FIN_1}\ G.\mathsf{gg\_finite}\ (\mathsf{pair}\ H_1\ U.\mathbf{2}))$

Together with the assumption H_2 it thus fulfils the axioms necessary to make it a Galois group.

▷ Prove $\mathsf{FT_1i} : U.\mathsf{is\_galois\_group}$
$= \mathsf{pair}\ \mathsf{P_1}\ H_2$

10.3.2.1.2 Proof of FT_1ii

The second result is also immediate; G^{∇} is a Galois subfield since it is (equal to) the fixed subfield of the Galois group G.

▷ Prove $\mathsf{FT}_1\mathsf{ii}$: $G^{\nabla}.\mathsf{is\_galois\_subfield}\ L$
= $(G, \mathsf{equal\_subs\_refl}\ G^{\nabla} : G^{\nabla}.\mathsf{is\_galois\_subfield}\ L)$

▷ Discharge $\mathsf{P}_1, H_2, H_1, U$

10.3.2.1.3 Proof of FT_1iii

For the third result, I have already that

▷ $\mathsf{GC}_3\ G : G \subseteq G^{\nabla\Delta}$

and wish to prove the reverse inclusion also. The two subsets are subgroups of $\mathsf{Aut}\ L$, but I prove the inclusion via intervening subsets of $\mathsf{map}\ F\ F$ and lift the result at the end.

▷ We want to prove subresult P_2
: $G^{\nabla\Delta}.\mathsf{as\_a\_subset\_of\_maps} \subseteq G.\mathsf{as\_a\_subset\_of\_maps}$

From comparing axioms, every element of $G^{\nabla\Delta}$ is a G^{∇}-fixing L-algebra morphism.

▷ Prove subresult $\mathsf{P}_2\mathsf{a} : G^{\nabla\Delta}.\mathsf{as\_a\_subset\_of\_maps} \subseteq (G^{\nabla}\text{-}\mathsf{fix\_hom}\ L)$
= $\mathsf{ALG}_4\ L\ G^{\nabla}$

But the corollary to $\mathsf{THM\_B}_1$ shows that this set is in turn equal to G restricted to L.

▷ Prove subresult $\mathsf{P}_2\mathsf{b} : (G^{\nabla}\text{-}\mathsf{fix\_hom}\ L) \subseteq (G{\restriction}L)$
= $(\mathsf{THM\_B}_4\ G).\mathsf{snd}$

Now $G \subseteq (\mathsf{Perm}\ L)$, and so restrictions to L have no effect.

▷ Prove subresult $\mathsf{P}_2\mathsf{c'} : \{\forall g : G\}\ g = (g{\restriction}L)$
= $[\lambda g : G]\ (\mathsf{REST}_3\ L\ g).\mathsf{fst}\ (G.\mathbf{1}.\mathbf{2}\ g).\mathsf{fst}$

▷ Locally construct by refinement $\mathsf{P}_2\mathsf{c} : (G{\restriction}L) \subseteq G.\mathsf{as\_a\_subset\_of\_maps}$
(map, refl, iso, rep_map, =, P_2c', $\restriction$, as_a_subset_of_maps, eq_closed,
restriction_perm, perm, AA_4, apply_across, $\subseteq$, $\in$, snd)

Thus the inclusion is proved:

▷ Prove subresult as claimed P_2
: $G^{\nabla\triangle}$.as_a_subset_of_maps $\subseteq$ G.as_a_subset_of_maps
= P_2a.subs_tran (P_2b.subs_tran P_2c)

Since equality in map F F and equality in Aut L are the same relation, I can lift the inclusion to one between subgroups of Aut L.

▷ Locally construct by refinement P_3 : $G^{\nabla\triangle} \subseteq G$
(P_2, $^{\nabla}$, $^{\triangle}$, iso, map, rep_map, =, AA_2s')

Hence the two subgroups are proved to be equal.

▷ Prove FT_1iii : $G = G^{\nabla\triangle}$
= pair (GC$_3$ G) P_3

▷ Discharge $\mathsf{P}_3, \mathsf{P}_2, \mathsf{P}_2$c, P_2c', P_2b, P_2a

▷ Discharge to prove as claimed FT_1
: $\{\forall G$: galois_group $L\}$ $[\delta \mathsf{ft}_1\mathsf{i} = \{\forall U$: subgroup_of $G\}$
(U.is_decideable_in G) $\rightarrow$ (L.is_fin_dim_over U^{∇}) $\rightarrow$ U.is_galois_group]
$[\delta \mathsf{ft}_1\mathsf{ii} = G^{\nabla}$.is_galois_subfield $L]$ $[\delta \mathsf{ft}_1\mathsf{iii} = G = G^{\nabla\triangle}]$ $\bigwedge_3$ ft_1i ft_1ii ft_1iii
using
pair$_3$' FT_1i FT_1ii FT_1iii
: $\bigwedge_3$ ($\{\forall U \mid$ subgroup_of $G\}$ (U.is_decideable_in G) $\rightarrow$ (L.is_fin_dim_over U^{∇}) $\rightarrow$
U.is_galois_group) (G^{∇}.is_galois_subfield L) ($G = G^{\nabla\triangle}$)
▷ Discharge G

10.3.2.2 Proof of second part of the fundamental theorem

▷ Introduce K : galois_subfield L

▷ Let $G = K$.gf_gg : galois_group L and Q = K.gf_equal : $G^{\nabla} = K$

For this part of the fundamental theorem, most of the work goes into proving the first result.

10.3.2.2.1 Proof of $\mathbf{FT}_2$i

▷ Introduce J : K-subfield

THM_B$_3$ gives me an important equality between J and $J^{\triangle\nabla}$.

▷ Prove subresult Q' : $J = J^{\Delta\nabla}$
$= \mathsf{THM\_B_3}\ G\ J$

Thus J is a Galois subfield if J^Δ is a Galois subgroup. I shall use $\mathsf{FT_1i}$ to prove this. I need to show that $J^\Delta \subseteq G$ and that it fulfills the two hypotheses:

▷ We want to prove subresult $\mathsf{H_1}$: J^Δ.is_decideable_in G

▷ We want to prove subresult $\mathsf{H_2}$: L.is_fin_dim_over $J^{\Delta\nabla}$

Firstly, since $K \subseteq J$, I have that $J^\Delta \subseteq K^\Delta = G^{\nabla\Delta}$. Since

▷ $\mathsf{FT_1iii}\ G$: $G = G^{\nabla\Delta}$

this means that

▷ J^Δ : subgroup_of G

as required.

Hypothesis H_1

I now prove the first hypothesis H_1.

▷ Introduce g : G

Now $g \in J^\Delta$ if and only if it is equal to the identity on J.

▷ $\mathsf{P_2}$: $(g.(=\restriction J)\ \mathsf{id}) \leftrightarrow (g \in J^\Delta)$

A useful subresult $\mathsf{SPAN_4}$ shows that if G^∇ is finitely generated by J, this equality of maps can be decided under certain conditions. (This result will be proved in the next chapter as an example of a proof at a lower level.) Since g is an automorphism on L, it is clearly an homomorphism on $J \subseteq L$, as is the identity map.

▷ $(\mathsf{Hg}, \mathsf{Hi})$: $(g \in (\mathsf{subgroup\_hom}\ J)) \wedge (\mathsf{id} \in (\mathsf{subgroup\_hom}\ J))$

Now, let A : subset F be the subset where those two homomorphisms agree; that is, everything fixed by g.

▷ $x \in \mathsf{A} \simeq g.\mathsf{fixes}\ x$

Now $\mathsf{SPAN_4}$ will establish the required decideability result:

▷ SPAN_4 : $(G^\nabla \subseteq J) \to (J.\mathsf{has\_fin\_span\_over}\ G^\nabla) \to$
$(\{\forall v : J \cap \mathsf{A}\}\ \{\forall x : G^\nabla\}\ (x \times v) \in \mathsf{A}) \to (J \subseteq \mathsf{A}).\mathsf{or\_not}$

The first two conditions are trivially satisfied since J is by definition a G^∇-subfield.

▷ Prove subresult $\mathsf{P}_3\mathsf{a}$: $G^\nabla \subseteq J$
= Q.fst.subs_tran (subs_sub J)

▷ Prove subresult $\mathsf{P}_3\mathsf{b}$: $J.\mathsf{has\_fin\_span\_over}\ G^\nabla$
= span_fin_dim (FINDIM$_2$ J Q.equal_subs_symm (sub_fin_dim_over K J))

To apply SPAN_4 I need to prove the third condition also.

▷ We want to prove subresult $\mathsf{P}_3\mathsf{c}$: $\{\forall v : J \cap \mathsf{A}\}\ \{\forall x : G^\nabla\}\ (x \times v) \in \mathsf{A}$

▷ Introduce v : $J \cap \mathsf{A}$ and x : G^∇

I must prove that $x \times v$ is fixed by g. Now x and v are both in L, and g is an automorphism of L. This proves

▷ Locally construct by refinement Q_1 : $(g\ (x \times v)) = ((g\ x) \times (g\ v))$
(A, ∩, ev, ∈, fst, sub_subs, rep_map, ap, make, G, $^\nabla$, resp_plus_times,
as_a_subset_of_perm, perm, Perm, snd, ×, resp_map$_2$, iso, map, rep, +)

But $x \in G^\nabla$ and $v \in \mathsf{A}$, so g fixes both these elements.

▷ Locally construct by refinement Q_2 : $((g\ x) \times (g\ v)) = (x \times v)$
(A, ∩, ev, ∈, snd, G, $^\nabla$, fixes, fst, times_resp)

▷ Discharge to prove subresult as claimed $\mathsf{P}_3\mathsf{c}$
: $\{\forall v : J \cap \mathsf{A}\}\ \{\forall x : G^\nabla\}\ (x \times v) \in \mathsf{A}$
using
Q_1.tran Q_2 : g.fixes $(x \times v)$

▷ Discharge x, v

▷ Prove subresult P_3 : $(g.(=\restriction J)\ \mathsf{id}).\mathsf{or\_not}$
= SPAN_4 $\mathsf{P}_3\mathsf{a}$ $\mathsf{P}_3\mathsf{b}$ $\mathsf{P}_3\mathsf{c}$

▷ Discharge to prove subresult as claimed H_1 : $J^\Delta.\mathsf{is\_decideable\_in}\ G$
using
DEC_2 P_2 P_3 : $(g \in J^\Delta).\mathsf{or\_not}$

▷ Discharge g

Hypothesis H_2

This is immediate since L has finite dimension over the K-subfield J, and I already have from $\mathsf{THM\_B}_3$ that $J = J^{\Delta\nabla}$.

▷ Prove subresult as claimed H_2 : L.is_fin_dim_over $J^{\Delta\nabla}$
= FINDIM$_2$ L Q' (fin_dim_over_sub J)

Therefore by FT_1i, J^{Δ} is a Galois group

▷ Prove subresult P_4 : J^{Δ}.is_galois_group
= FT$_1$i $G|J^{\Delta}$ H$_1$ H$_2$

and the result that J is a Galois subfield then follows, since by Q' it is equal to the fixed subfield of this Galois group.

▷ Construct by refinement FT_2i : J.is_galois_subfield L
(Q', $^{\Delta}$, $^{\nabla}$, equal_subs_symm, perm, equal_subs_refl, P$_4$, GC$_1$eq, equal_subs_tran, =, galois_group)

▷ Discharge $\mathsf{P}_4, \mathsf{H}_2, \mathsf{H}_1, \mathsf{P}_3, \mathsf{P}_3\mathsf{c}, \mathsf{Q}_2, \mathsf{Q}_1, \mathsf{P}_3\mathsf{b}, \mathsf{P}_3\mathsf{a}, \mathsf{A}, \mathsf{Hi}, \mathsf{Hg}, \mathsf{P}_2, \mathsf{Q}', J$

10.3.2.2.2 Proof of $\mathbf{FT_2ii}$

I need to prove that K^{Δ} is a Galois group. This is easy as FT_1iii shows that $G = G^{\nabla\Delta} = K^{\Delta}$.

▷ Prove subresult Q_3 : $G = K^{\Delta}$
= (FT$_1$iii G).equal_subs_tran (GC$_2$eq Q)

Then since G is a Galois group, so is K^{Δ}.

▷ Prove FT_2ii : K^{Δ}.is_galois_group
= GAL$_1$ Q$_3$ G.**2**

10.3.2.2.3 Proof of $\mathbf{FT_2iii}$

Applying the fix operator to both sides of the same equality Q_3 proves the third result.

▷ GC$_1$eq Q$_3$: $G^{\nabla} = K^{\Delta\nabla}$

▷ Prove FT_2iii : $K = K^{\Delta\nabla}$
= Q.equal_subs_symm.equal_subs_tran (GC$_1$eq Q$_3$)

▷ Discharge $\mathsf{Q}_3, \mathsf{Q}, G$

▷ Discharge to prove as claimed FT_2
: $\{\forall K : \mathsf{galois\_subfield}\ L\}\ [\delta\mathsf{ft}_2\mathsf{i} = \{\forall J : K\text{-}\mathsf{subfield}\}\ J.\mathsf{is\_galois\_subfield}\ L]$
$[\delta\mathsf{ft}_2\mathsf{ii} = K^{\triangle}.\mathsf{is\_galois\_group}]\ [\delta\mathsf{ft}_2\mathsf{iii} = K = K^{\triangle\triangledown}]\ \bigwedge_3\ \mathsf{ft}_2\mathsf{i}\ \mathsf{ft}_2\mathsf{ii}\ \mathsf{ft}_2\mathsf{iii}$
using
$\mathsf{pair}_3'\ \mathsf{FT}_2\mathsf{i}\ \mathsf{FT}_2\mathsf{ii}\ \mathsf{FT}_2\mathsf{iii}$
: $\bigwedge_3\ (\{\forall J : K\text{-}\mathsf{subfield}\}\ J.\mathsf{is\_galois\_subfield}\ L)\ K^{\triangle}.\mathsf{is\_galois\_group}\ (K = K^{\triangle\triangledown})$
▷ Discharge K

10.3.2.3 Lemma 2

▷ Introduce K : galois_subfield L; let $G = K^{\triangle}$: subgroup_of (Aut $L_{.\mathbf{1}}$) and
introduce J : K-subfield

10.3.2.3.1 Statement of Lemma 2

The statement of this lemma is based around the logical equivalence of two propositions:

▷ Let $\mathsf{p}_1 = J^{\triangle}.\mathsf{normal\_in}\ G$: prop

▷ Let $\mathsf{p}_2 = (G{\restriction}J) \subseteq (\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps}$: prop

▷ $\mathsf{p}_1 \leftrightarrow \mathsf{p}_2$: prop

When I use the lemma, I will want slightly different versions of p_2 in it.

▷ We want to prove $\mathsf{LEMMA}_2\mathsf{i}$
: $(J^{\triangle}.\mathsf{normal\_in}\ G) \rightarrow (G{\restriction}J) = (\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps}$

▷ We want to prove $\mathsf{LEMMA}_2\mathsf{ii}$: $((G{\restriction}J) \cong (\mathsf{aut}\ J\ K)) \rightarrow J^{\triangle}.\mathsf{normal\_in}\ G$

10.3.2.3.2 Preliminaries

I start by noting a subset inclusion that will be used a couple of times. From the results $\mathsf{FT}_2\mathsf{ii}$ and $\mathsf{FT}_2\mathsf{iii}$ I previously proved, I know that G is a Galois group, and J is a $G^{\triangledown}$-subfield. By reordering axioms and using $\mathsf{THM\_B}_1$, I have that

▷ P_0 : $(\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps} \subseteq (G{\restriction}J)$

Also note that J is a Galois subfield and hence is equal to the fixed subfield of its fixing group.

▷ Prove subresult $\mathsf{Q} : J = J^{\Delta\nabla}$
$= \mathsf{FT_2iii}\ (\mathsf{FT_2i}\ K\ J)$

I split the logical equivalence $\mathsf{p}_1 \leftrightarrow \mathsf{p}_2$ into four separate parts via an intermediate proposition:

▷ Let $\mathsf{p} = \{\forall g : G\}\ \{\forall x : J\}\ (g\ x) \in J^{\Delta\nabla} : \mathsf{prop}$

10.3.2.3.3 The first two implications, $\mathsf{p}_1 \leftrightarrow \mathsf{p}$

The first implication will be useful in its own right later in the development, so I name it as a third part to this lemma.

▷ We want to prove $\mathsf{LEMMA_2iii}$
$: (J^{\Delta}.\mathsf{normal\_in}\ G) \rightarrow \{\forall g : G\}\ \{\forall x : J\}\ (g\ x) \in J^{\Delta\nabla}$

That p_1 and p are equivalent statements follows from definitions and some equational reasoning.

▷ $\mathsf{p}_1 \simeq \{\forall g : G\}\ \{\forall h : J^{\Delta}\}\ (g^{-1} \circ (h \circ g)) \in J^{\Delta}$

▷ Introduce $g : G$ and $h : J^{\Delta}$

Recall that for any $f : \mathsf{perm}\ F$

▷ $f \in J^{\Delta} \simeq (\{\forall x : J\}\ f.\mathsf{fixes}\ x) \wedge (f \in (\mathsf{Aut}\ L))$

Now g, h, and their combinations are known to be elements of $\mathsf{Aut}\ L$, so I need concentrate only on their fixing behaviour.

▷ Introduce $x : J$

A little equational reasoning shows that

▷ $(\mathsf{P_1}', \mathsf{P_2}') : ((g^{-1} \circ (h \circ g)).\mathsf{fixes}\ x) \leftrightarrow (h.\mathsf{fixes}\ (g\ x))$

Now by definition,

▷ $(g\ x) \in J^{\Delta\nabla} \simeq (\{\forall h : J^{\Delta}\}\ h.\mathsf{fixes}\ (g\ x)) \wedge ((g\ x) \in L)$

Since $g\ x$ is always in L, this equivalence of fixings means I have proved my result:

▷ $(\mathsf{P_1}, \mathsf{P_2}) : \mathsf{p}_1 \leftrightarrow \mathsf{p}$

▷ Prove as claimed $\mathsf{LEMMA_2iii}$
: $(J^{\triangle}.\mathsf{normal\_in}\ G) \rightarrow \{\forall g : G\}\ \{\forall x : J\}\ (g\ x) \in J^{\triangle\triangledown}$
$= \mathsf{P_1}$

▷ Discharge $\mathsf{P_2'}, \mathsf{P_1'}, x, f, h, g$

10.3.2.3.4 The third implication, $\mathsf{p_2} \rightarrow \mathsf{p}$

▷ Suppose H : $\mathsf{p_2}$

I want to prove the intermediate proposition p.

▷ Introduce g : G and x : J

Now $(g\ x) = (g{\restriction}J\ x)$. Since, by the hypothesis H, $g{\restriction}J$ is an automorphism of J, it is closed on J. Hence

▷ $\mathsf{P_3'}$: $(g\ x) \in J$

▷ Prove subresult $\mathsf{P_3}$: $(g\ x) \in J^{\triangle\triangledown}$
$= \mathsf{Q.fst\ P_3'}$

▷ Discharge x, g, H

10.3.2.3.5 The fourth implication, $\mathsf{p} \rightarrow \mathsf{p_2}$

▷ Suppose H : p

I want to prove the proposition $\mathsf{p_2}$. I will use a little lemma about the restriction of a function and its inverse.

▷ Introduce g_1, g_2 : $\mathsf{perm}\ F$; suppose P_g_2 : $g_2 \in G$ and Qg : $(g_1 \circ g_2) = \mathsf{id}$

▷ We want to prove subresult $\mathsf{little\_lemma}$: $((g_1{\restriction}J) \circ (g_2{\restriction}J)) = \mathsf{id}$

▷ Introduce x : F

▷ Suppose Hn : $x \notin J$

Any restriction to J will not touch x, so clearly

▷ Cn : $((g_1{\restriction}J) \circ (g_2{\restriction}J)).\mathsf{fixes}\ x$

▷ Suppose Hy : $x \in J$

I have that

▷ $(\mathsf{REST_1}\ J\ g_2\ Hy).\mathsf{symm}$: $(g_2{\restriction}J\ x) = (g_2\ x)$

▷ $H\ P\_g_2\ Hy$: $(g_2\ x) \in J^{\Delta\nabla}$

Therefore using Q I have that $(g_2\ x) \in J$ and hence $g_1{\restriction}J$ has the same effect on $g_2\ x$ that g_1 does. Thus

▷ Cy : $((g_1{\restriction}J) \circ (g_2{\restriction}J))$.fixes x

▷ Discharge Hy, Hn

▷ Discharge to prove subresult as claimed little_lemma : $((g_1{\restriction}J) \circ (g_2{\restriction}J)) = \mathsf{id}$
using
case $(x \in_?\ J)$ Cy Cn : $((g_1{\restriction}J) \circ (g_2{\restriction}J))$.fixes x

▷ Discharge x

▷ Discharge Qg, P_g_2, g_2, g_1

I want to prove that $(G{\restriction}J) \subseteq (\mathsf{aut}\ J\ K)$.as_a_subset_of_maps.

▷ Introduce g : $G{\restriction}J$ and let $g' = (g.\mathsf{ev}).\mathbf{1}.\mathsf{rep}$: perm F

The little_lemma proves that $g = (g'{\restriction}J)$ has $g'^{-1}{\restriction}J$ as its inverse and is hence an permutation of F. It fixes $K \subseteq J$ because g' does, and because $g.(=\!{\restriction}J)\ g'$. For the same reason it also acts as an automorphism on J. Therefore

▷ P_4 : $g \in (\mathsf{aut}\ J\ K)$.as_a_subset_of_maps

▷ Discharge g, H

10.3.2.3.6 The end of the proof

▷ Prove as claimed $\mathsf{LEMMA}_2\mathsf{i}$
: $(J^{\Delta}.\mathsf{normal\_in}\ G) \to (G{\restriction}J) = (\mathsf{aut}\ J\ K)$.as_a_subset_of_maps
$= [\lambda H : \mathsf{p}_1]\ \mathsf{pair}\ (\mathsf{P}_4\ (\mathsf{P}_1\ H))\ \mathsf{P}_0$

▷ Suppose H : $(G{\restriction}J) \cong (\mathsf{aut}\ J\ K)$

The result $\mathsf{THM\_B}_0$ gives me that $G{\restriction}J$ is finite. Its size is equal to that of (aut $J\ K) \cong (\mathsf{aut}\ J\ K)$.as_a_subset_of_maps. Since by P_0 the latter is a subset of $G{\restriction}J$, it follows that these two subsets are in fact equal.

▷ H' : $(\mathsf{aut}\ J\ K)$.as_a_subset_of_maps $= (G{\restriction}J)$

The second part of the lemma then follows using on H' the other two implications that were proved earlier.

▷ Discharge to prove as claimed $\mathsf{LEMMA_2ii}$
: $((G\restriction J) \cong (\mathsf{aut}\ J\ K)) \to J^{\triangle}.\mathsf{normal\_in}\ G$

using

$\mathsf{P_2}\ (\mathsf{P_3}\ \mathsf{H'.snd})$: $\mathsf{p_1}$

▷ Discharge H

▷ Discharge J, G, K

10.3.2.4 Proof of final part of the fundamental theorem

▷ Introduce K : $\mathsf{galois\_subfield}\ L$ and J : K-$\mathsf{subfield}$

For the first two parts of this final result, I need to show that $K.\mathsf{is\_galois\_subfield}$ J if and only if $J^{\triangle}.\mathsf{normal\_in}\ K^{\triangle}$.

10.3.2.4.1 Proof of $\mathbf{FT_3i}$

▷ Suppose H : $K.\mathsf{is\_galois\_subfield}\ J$

▷ We want to prove $\mathsf{FT_3i}$: $J^{\triangle}.\mathsf{normal\_in}\ K^{\triangle}$

From $\mathsf{LEMMA_2ii}$, it will suffice to prove that $(K^{\triangle}\restriction J) \cong (\mathsf{aut}\ J\ K)$.

Now,

▷ $\mathsf{FT_2ii}\ H$: $(\mathsf{aut}\ J\ K).(\mathsf{is\_galois\_group}|J)$

and so by the corollary $\mathsf{THM\_A_3}$ (with J replacing L) applied to this Galois group, I have

▷ $\mathsf{P_1}$: $\langle\exists n : \mathbb{N}\rangle\ \langle\exists \boldsymbol{w} : F^{\wedge}n\rangle$

$(J.\mathsf{has\_basis\_over}\ (\mathsf{fix}\ J\ (\mathsf{aut}\ J\ K))\ \boldsymbol{w}) \wedge ((\mathsf{aut}\ J\ K) \cong \boldsymbol{w})$

But since J is already finite dimensional over K, and since

▷ $\mathsf{FT_2iii}\ H$: $K = (\mathsf{fix}\ J\ (\mathsf{aut}\ J\ K))$

I have that

▷ $\mathsf{P_2}$: $\langle\exists m : \mathbb{N}\rangle\ \langle\exists \boldsymbol{v} : F^{\wedge}m\rangle\ J.\mathsf{has\_basis\_over}\ (\mathsf{fix}\ J\ (\mathsf{aut}\ J\ K))\ \boldsymbol{v}$

Therefore these bases have the same size:

▷ $\mathsf{Q_1}$: $\boldsymbol{v} \cong \boldsymbol{w}$

But by $\mathsf{THM\_B}_2$,

▷ $\mathsf{Q}_2 : (K^{\Delta}\restriction J) \cong \boldsymbol{v}$

which completes the proof that $K^{\Delta}\restriction J$ has the same size as $\mathsf{aut}\ J\ K$. Hence the required result follows.

▷ Prove as claimed $\mathsf{FT}_3\mathsf{i} : J^{\Delta}.\mathsf{normal\_in}\ K^{\Delta}$
$= \mathsf{LEMMA}_2\mathsf{ii}\ K\ J\ ((\mathsf{Q}_2.\mathsf{eqsize\_tran}\ \mathsf{Q}_1).\mathsf{eqsize\_tran}\ \mathsf{P}_1.\mathbf{2}.\mathbf{2}.\mathsf{snd}.\mathsf{eqsize\_symm})$

▷ Discharge $\mathsf{Q}_1, \mathsf{Q}_2, \boldsymbol{v}, \mathsf{m}, \boldsymbol{w}, \mathsf{n}, \mathsf{P}_2, \mathsf{P}_1$

▷ Discharge H

10.3.2.4.2 Proof of $\mathbf{FT}_3$ii

▷ Suppose $H : J^{\Delta}.\mathsf{normal\_in}\ K^{\Delta}$

▷ We want to prove $\mathsf{FT}_3\mathsf{ii} : K.\mathsf{is\_galois\_subfield}\ J$

From the hypothesis I have that

▷ Prove subresult $\mathsf{P}_1 : (K^{\Delta}\restriction J) = (\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps}$
$= \mathsf{LEMMA}_2\mathsf{i}\ K\ J\ H$

I shall prove that this subgroup has a fixed subfield equal to K and is also a Galois group on J.

I already know that

▷ $\mathsf{GC}_4\ (\mathsf{subs\_sub}\ J) : K \subseteq (\mathsf{fix}\ J\ (\mathsf{aut}\ J\ K))$

and need to prove the reverse inclusion. Now

▷ $\mathsf{FT}_2\mathsf{iii}\ K : K = K^{\Delta\nabla}$

I need to show that $(\mathsf{fix}\ J\ (\mathsf{aut}\ J\ K)) \subseteq K^{\Delta\nabla} = (\mathsf{fix}\ L\ (\mathsf{aut}\ L\ K))$.

▷ Introduce $x : \mathsf{fix}\ J\ (\mathsf{aut}\ J\ K)$ and $g : K^{\Delta}$

Now $x \in J \subseteq L$. Also, $(g\restriction J) \in (K^{\Delta}\restriction J) = \mathsf{aut}\ J\ K$. Therefore

▷ $\mathsf{P}_3' : (g\restriction J).\mathsf{fixes}\ x$

and then since $(g\ x) = (g\restriction J\ x)$, I have that $g.\mathsf{fixes}\ x$ also, completing the proof.

▷ Discharge g, x

▷ $\mathsf{P}_3 : K = (\mathsf{fix}\ J\ (\mathsf{aut}\ J\ K))$

It remains for me to show that aut J K is a galois group on J. Using the above equality, I have that

▷ Prove subresult $\mathsf{P}_{2\text{-}2}$: J.is_fin_dim_over (fix J (aut J K))
= FINDIM_2 J P_3 (sub_fin_dim_over K J)

and the group is finite because it is equal in size to $K^{\Delta}\restriction J$, which is finite by $\mathsf{THM\_B}_0$.

▷ $\mathsf{P}_{2\text{-}1}$: (aut J K).is_finite

Therefore

▷ Prove subresult P_2 : (aut J K).is_galois_group
= pair $\mathsf{P}_{2\text{-}1}$ $\mathsf{P}_{2\text{-}2}$

▷ Prove as claimed FT_3ii : K.is_galois_subfield J
= (P_2, P_3.equal_subs_symm : K.is_galois_subfield J)

▷ Discharge $\mathsf{P}_2, \mathsf{P}_{2\text{-}2}, \mathsf{P}_{2\text{-}1}, \mathsf{P}_3$

▷ Discharge P_1, H

10.3.2.4.3 Proof of $\mathbf{FT}_3$iii

I need to exhibit an isomorphism between the groups K^{Δ}/J^{Δ} and aut J K. I shall build this isomorphism in four stages.

▷ Let $U = K^{\Delta} \cap J^{\Delta}$: subgroup (perm F)

▷ Suppose H : K.is_galois_subfield J

Since

▷ GC_2 (subs_sub J) : $J^{\Delta} \subseteq K^{\Delta}$

it follows that $J^{\Delta} = U$. Hence there is an identity isomorphism

▷ π_1 : iso (K^{Δ}/J^{Δ}) (K^{Δ}/U)

Now by LEMMA_1ii the restriction to J forms an isomorphism,

▷ π_2 : iso (K^{Δ}/U) $(K^{\Delta}\restriction J)$

Under the hypothesis H,

▷ LEMMA_2i K J (FT_3i H) : $(K^{\Delta}\restriction J)$ = (aut J K).as_a_subset_of_maps

Therefore there is an identity isomorphism,

▷ π_3 : iso $(K^{\Delta}\restriction J)$ (aut J K).as_a_subset_of_maps

Finally, considering this subset of maps as a subgroup of field automorphisms has no computational effect on it:

▷ π_4 : iso (aut J K).as_a_subset_of_maps (aut J K)

Composing these (mostly identity-like) parts gives me my isomorphism,

▷ Let $\pi = ((\pi_4 \circ \pi_3) \circ \pi_2) \circ \pi_1$: iso (K^{Δ}/J^{Δ}) (aut J K)

It remains for me to show that π has the required properties.

▷ Introduce $g, h : G/J^{\Delta}$ and $x : F$

The only computationally relevant part of this isomorphism is the restriction to J performed by π_2.

▷ P_2.symm : $(\pi\ g) = (g\restriction J)$

By LEMMA_2iii on the hypothesis H, g and h applied to arguments in J produce results in $J^{\Delta\nabla} = J$. Thus they are closed on J.

▷ P_3 : $(x \in J) \to (g\ x) \in J$

Therefore, restriction to J will distribute through composition of the two functions. This shows that π is an isomorphism of groups.

▷ Discharge x

▷ P_4 : $((h \circ g)\restriction J) = ((h\restriction J) \circ (g\restriction J))$

▷ Discharge h, g

▷ Prove FT_3iii : $\langle \exists \pi : \text{iso}\ (K^{\Delta}/J^{\Delta})\ (\text{aut}\ J\ K)\rangle\ (\{\forall g : K^{\Delta}/J^{\Delta}\}\ (g\restriction J) = (\pi\ g)) \wedge (\{\forall g, h : K^{\Delta}/J^{\Delta}\}\ ((h \circ g)\restriction J) = ((h\restriction J) \circ (g\restriction J)))$

$= (\pi, \text{pair}\ \mathsf{P}_2\ \mathsf{P}_4 : \langle \exists \pi : \text{iso}\ (K^{\Delta}/J^{\Delta})\ (\text{aut}\ J\ K)\rangle\ (\{\forall g : K^{\Delta}/J^{\Delta}\}\ (g\restriction J) = (\pi\ g))$
$\wedge\ (\{\forall g, h : K^{\Delta}/J^{\Delta}\}\ ((h \circ g)\restriction J) = ((h\restriction J) \circ (g\restriction J))))$

▷ Discharge H, U

▷ Discharge to prove as claimed FT_3

$\mathbf{:} \{\forall L \,|\, \mathsf{dsubfield}\ F\}\ \{\forall K : \mathsf{galois\_subfield}\ L\}\ \{\forall J : K\text{-}\mathsf{subfield}\}$
$[\delta \mathsf{ft}_3\mathsf{A} = K.\mathsf{is\_galois\_subfield}\ J]\ [\delta \mathsf{ft}_3\mathsf{B} = J^{\triangle}.\mathsf{normal\_in}\ K^{\triangle}]\ (\mathsf{ft}_3\mathsf{A} \leftrightarrow \mathsf{ft}_3\mathsf{B}) \wedge$
$(\mathsf{ft}_3\mathsf{A} \rightarrow \langle \exists \pi : \mathsf{iso}\ (K^{\triangle}/J^{\triangle})\ (\mathsf{aut}\ J\ K)\rangle\ (\{\forall g : K^{\triangle}/J^{\triangle}\}\ (g{\restriction}J) = (\pi\ g)) \wedge$
$(\{\forall g, h : K^{\triangle}/J^{\triangle}\}\ ((h \circ g){\restriction}J) = ((h{\restriction}J) \circ (g{\restriction}J))))$

using

$\mathsf{pair}_3\text{'}\ \mathsf{FT}_3\mathsf{i}\ \mathsf{FT}_3\mathsf{ii}\ \mathsf{FT}_3\mathsf{iii} : \bigwedge_3 ((K.\mathsf{is\_galois\_subfield}\ J) \rightarrow J^{\triangle}.\mathsf{normal\_in}\ K^{\triangle})$
$((J^{\triangle}.\mathsf{normal\_in}\ K^{\triangle}) \rightarrow K.\mathsf{is\_galois\_subfield}\ J)\ ((K.\mathsf{is\_galois\_subfield}\ J) \rightarrow$
$\langle \exists \pi : \mathsf{iso}\ (K^{\triangle}/J^{\triangle})\ (\mathsf{aut}\ J\ K)\rangle\ (\{\forall g : K^{\triangle}/J^{\triangle}\}\ (g{\restriction}J) = (\pi\ g)) \wedge$
$(\{\forall g, h : K^{\triangle}/J^{\triangle}\}\ ((h \circ g){\restriction}J) = ((h{\restriction}J) \circ (g{\restriction}J))))$

▷ Discharge J, K, L

▷ Discharge F

Chapter 11

A short detailed proof

Chapter 10 showed a formal proof at a relatively high-level compared to the detail required by the proof-checker. This short chapter presents a subresult at this lower level with all necessary details made explicit, so that the reader can see what this sort of presentation can look like.

11.1 A result about finite spans

I take as my example a result SPAN_4 which was used in proving the second third of the fundamental theorem. It also had application in order to prove a result about the decideability of a subset that was needed in the proof of theorem B.

▷ Introduce F | field and J, K | subfield F

11.1.1 Preliminary material

The statement of the result I shall use as my example involves two definitions that the reader has not yet seen, and its proof will require four subresults. I review these items now.

I define the homomorphisms of a subgroup:

▷ subgroup_hom J : subset (map F F)

These are mappings which respect binary addition within the subgroup J. By induction, this respectfulness extends to the action of these mappings on linear sums; this is the result LINSUM_2.

▷ Introduce g : subgroup_hom J and $n : \mathbb{N}$

▷ $g \in (\text{subgroup\_hom } J) \simeq \{\forall x, y : J\}\ (g\ (x + y)) = ((g\ x) + (g\ y))$

▷ $\mathsf{LINSUM}_2\ J\ g : \{\forall \boldsymbol{x} : J\hat{}n\}\ (g\ (\sum \boldsymbol{x})) = (\sum (g \circ \boldsymbol{x}))$

I should explain the use of composition above: recall that a tuple $\boldsymbol{x} : F\hat{}n$ is just an indexing map from n to F. Therefore the composition $g \circ \boldsymbol{x}$ is also a tuple, with

▷ $(g \circ \boldsymbol{x})\backslash i \simeq g\ (\boldsymbol{x}\backslash i)$

The other three subresults I shall require all involve decideability. First, recall that I only consider discrete fields. Secondly, the decideability of a proposition is preserved by logical equivalence. Finally the subresult FIN_6 is a formulation of the fact that a finite conjunction of decideable propositions is decideable; this is easily proved by induction.

▷ FIELD_1 : F.is_discrete

▷ DEC_2 : $\{\forall p_1, p_2\}\ (p_1 \leftrightarrow p_2) \to p_1$.or_not $\to p_2$.or_not

▷ FIN_6

: $\{\forall P : \text{subset } n\}\ (\{\forall i : n\}\ (i \in P) \vee (i \notin P)) \to (\{\forall i : n\}\ i \in P)$.or_not

▷ Discharge n, g

A final definition is that of the subset of some domain set upon which two mappings agree.

▷ Introduce $g_1, g_2 \mid$ subgroup_hom J

▷ Let $\mathsf{A} = g_1$.agree g_2 : subset F and introduce $x : F$

▷ $x \in \mathsf{A} \simeq (g_1\ x) = (g_2\ x)$

▷ Discharge x

11.1.2 Statement of the result

Under three conditions, two on the relationship of the subfields J and K, and one involving closedness on A,

▷ Suppose H_1 **:** $K \subseteq J$ and H_2 **:** $J.\mathsf{has\_fin\_span\_over}\ K$

▷ Suppose H_3 **:** $\{\forall w : J \cap \mathsf{A}\}\ \{\forall x : K\}\ (x * w) \in \mathsf{A}$

then it is decideable whether or not g_1 and g_2 agree across J.

▷ We want to prove SPAN_4 **:** $(J \subseteq \mathsf{A}).\mathsf{or\_not}$

11.1.3 The proof

Under the hypothesis H_2, I introduce names for the K-span of J, and the tuple that does the spanning.

▷ Let $\mathsf{n} = \mathsf{size\_of\_span}\ H_2$ **:** $\mathbb{N}$

▷ Let $\boldsymbol{w} = \mathsf{spanning\_tuple}\ H_2$ **:** $J\hat{}\mathsf{n}$

The proof decides the proposition $\boldsymbol{w} \in (\mathsf{A}\hat{}\mathsf{n})$ that the homomorphisms agree on each basis vector. This will be shown to be logically equivalent to agreement on all of J. The subresult DEC_2 then completes the proof.

▷ We want to prove subresult P_1 **:** $(\boldsymbol{w} \in (\mathsf{A}\hat{}\mathsf{n})) \vee (\boldsymbol{w} \notin (\mathsf{A}\hat{}\mathsf{n}))$

▷ We want to prove subresult P_2 **:** $(\boldsymbol{w} \in (\mathsf{A}\hat{}\mathsf{n})) \leftrightarrow (J \subseteq \mathsf{A})$

11.1.3.1 The deciding

I wish to apply the subresult FIN_6. Since the field is discrete, whether or not g_1 and g_2 agree on an individual spanning element $\boldsymbol{w}_\backslash i$ is decideable.

▷ Prove subresult $\mathsf{P}_{1\text{-}1}$ **:** $\{\forall x : J\}\ (x \in \mathsf{A}) \vee (x \notin \mathsf{A})$
$= [\lambda x : J]\ \mathsf{FIELD}_1\ (g_1\ x)\ (g_2\ x)$

▷ Prove subresult $\mathsf{P}_{1\text{-}2}$ **:** $\{\forall i : \mathsf{n}\}\ ((\boldsymbol{w}_\backslash i) \in \mathsf{A}) \vee ((\boldsymbol{w}_\backslash i) \notin \mathsf{A})$
$= [\lambda i : \mathsf{n}]\ \mathsf{P}_{1\text{-}1}\ (\mathsf{make}\ (\boldsymbol{w}.\mathsf{ev}\ i))$

In order to apply FIN_6, I also need to prove that the indices $i : \mathsf{n}$ such that $(\boldsymbol{w}_\backslash i) \in \mathsf{A}$ form a subset. This is an example of where the lack of an intensional

equality introduces a trivial but tedious extra proof step into the formalisation.

▷ Prove subresult $\mathsf{P}_{1\text{-}3}$: subset_axioms $([\lambda i : \mathsf{n}]\ (\boldsymbol{w}_{\backslash} i) \in \mathsf{A})$
$= [\lambda i_1, i_2 : \mathsf{n}]\ [\lambda Q i : i_1 = i_2]\ [\lambda H : (\boldsymbol{w}_{\backslash} i_1) \in \mathsf{A}]$ eq_closed $(\boldsymbol{w}.\mathsf{rep.resp}\ Q i)\ H$

▷ Let $\mathsf{P}_3 = ([\lambda i : \mathsf{n}]\ (\boldsymbol{w}_{\backslash} i) \in \mathsf{A}, \mathsf{P}_{1\text{-}3} : \mathsf{subset}\ \mathsf{n}) : \mathsf{subset}\ \mathsf{n}$

Therefore,

▷ Prove subresult as claimed $\mathsf{P}_1 : (\boldsymbol{w} \in (\mathsf{A}^{\wedge}\mathsf{n})) \vee (\boldsymbol{w} \notin (\mathsf{A}^{\wedge}\mathsf{n}))$
$= \mathsf{FIN}_6\ \mathsf{P}_3\ \mathsf{P}_{1\text{-}2}$

11.1.3.2 The logical equivalence

I need to prove a logical implication in both directions;

▷ We want to prove subresult $\mathsf{P}_{2\text{-}1} : (\boldsymbol{w} \in (\mathsf{A}^{\wedge}\mathsf{n})) \to J \subseteq \mathsf{A}$

▷ We want to prove subresult $\mathsf{P}_{2\text{-}2} : (J \subseteq \mathsf{A}) \to \boldsymbol{w} \in (\mathsf{A}^{\wedge}\mathsf{n})$

The reverse implication is trivial since if $J \subseteq \mathsf{A}$ then $\boldsymbol{w} \in (J^{\wedge}\mathsf{n}) \subseteq (\mathsf{A}^{\wedge}\mathsf{n})$.

▷ Suppose $H_4 : J \subseteq \mathsf{A}$

▷ Discharge to prove subresult as claimed $\mathsf{P}_{2\text{-}2} : (J \subseteq \mathsf{A}) \to \boldsymbol{w} \in (\mathsf{A}^{\wedge}\mathsf{n})$
using $[\lambda i : \mathsf{n}]\ H_4\ (\boldsymbol{w}.\mathsf{ev}\ i) : \boldsymbol{w} \in (\mathsf{A}^{\wedge}\mathsf{n})$

▷ Discharge H_4

It remains for me to prove the forward implication. This is the important part of the proof, which requires the hypotheses H_1, H_2 and H_3.

▷ Suppose $H_5 : \boldsymbol{w} \in (\mathsf{A}^{\wedge}\mathsf{n})$

▷ Introduce $v : J$

The proof proceeds by finding a linear combination over K for v.

▷ Prove subresult $\mathsf{P}_4 : v \in (\boldsymbol{w}\text{-span\_over}\ K)$
$= H_{2\cdot\mathbf{2}\cdot\mathbf{2}}.\mathsf{snd.fst}\ v$

▷ Let $\boldsymbol{x} = \mathsf{P}_{4\cdot\mathbf{1}} : K^{\wedge}\mathsf{n}$

I abbreviate the combination that sums to v:

▷ Let $\boldsymbol{v} = \boldsymbol{x} * \boldsymbol{w} : F^{\wedge}\mathsf{n}$

▷ Prove subresult $\mathsf{P}_5 : (\sum \boldsymbol{v}) = v$
$= \mathsf{P}_{4\cdot\mathbf{2}}$

Now since $\boldsymbol{x} \in (K^{\wedge}\mathsf{n}) \subseteq (J^{\wedge}\mathsf{n})$, and also $\boldsymbol{w} \in (J^{\wedge}\mathsf{n})$, so is their combination.

▷ Prove subresult $\mathsf{P}_6 : \boldsymbol{v} \in (J\hat{}\mathsf{n})$
$= [\lambda i : \mathsf{n}]\ \mathsf{times\_closed}\ (H_1\ (\boldsymbol{x}.\mathsf{ev}\ i))\ (\boldsymbol{w}.\mathsf{ev}\ i)$

LINSUM_2 shows that the applications of g_1 and g_2 to the linear combination can be moved through the summation sign.

▷ Prove subresult $\mathsf{P}_7 : \{\forall g : \mathsf{subgroup\_hom}\ J\}\ (g\ (\sum \boldsymbol{v})) = (\sum (g \circ \boldsymbol{v}))$
$= [\lambda g : \mathsf{subgroup\_hom}\ J]\ \mathsf{LINSUM}_2\ J\ g\ \mathsf{P}_6$

I have that $\boldsymbol{w}$ is in both $J\hat{}\mathsf{n}$ and $\mathsf{A}\hat{}\mathsf{n}$. Hence H_3 gives me that g_1 and g_2 agree on each tuple component $(\boldsymbol{w} * \boldsymbol{x})\backslash i$.

▷ Prove subresult $\mathsf{P}_8 : \boldsymbol{w} \in ((J \cap \mathsf{A})\hat{}\mathsf{n})$
$= [\lambda i : \mathsf{n}]\ \mathsf{pair}\ (\boldsymbol{w}.\mathsf{ev}\ i)\ (H_5\ i)$

▷ Prove subresult $\mathsf{P}_9 : (g_1 \circ \boldsymbol{v}) = (g_2 \circ \boldsymbol{v})$
$= [\lambda i : \mathsf{n}]\ H_3\ (\mathsf{P}_8\ i)\ (\boldsymbol{x}.\mathsf{ev}\ i)$

Combining this with P_7 shows that $g_1\ v = \sum (g_1 \circ \boldsymbol{v}) = \sum (g_2 \circ \boldsymbol{v}) = g_2\ v$, as was to be proved.

▷ Prove subresult $\mathsf{P}_{10} : (\sum \boldsymbol{v}) \in \mathsf{A}$
$= ((\mathsf{P}_7\ g_1).\mathsf{tran}\ (\textstyle\sum.\mathsf{resp}\ \mathsf{P}_9)).\mathsf{tran}\ (\mathsf{P}_7\ g_2).\mathsf{symm}$

▷ Discharge to prove subresult as claimed $\mathsf{P}_{2\_1} : (\boldsymbol{w} \in (\mathsf{A}\hat{}\mathsf{n})) \to J \subseteq \mathsf{A}$
using $\mathsf{eq\_closed}\ \mathsf{P}_5\ \mathsf{P}_{10} : v \in \mathsf{A}$

▷ Discharge v, H_5

11.1.3.3 Conclusion

The logical equivalence is thus proved

▷ Prove subresult as claimed $\mathsf{P}_2 : (\boldsymbol{w} \in (\mathsf{A}\hat{}\mathsf{n})) \leftrightarrow (J \subseteq \mathsf{A})$
$= \mathsf{pair}\ \mathsf{P}_{2\_1}\ \mathsf{P}_{2\_2}$

and the required result follows from applying DEC_2 to the two main subresults.

▷ Prove as claimed $\mathsf{SPAN}_4 : (J \subseteq \mathsf{A}).\mathsf{or\_not}$
$= \mathsf{DEC}_2\ \mathsf{P}_2\ \mathsf{P}_1$

▷ Discharge $\mathsf{P}_2, \mathsf{P}_{2\_1}, \mathsf{P}_{10}, \mathsf{P}_9, \mathsf{P}_8, \mathsf{P}_7, \mathsf{P}_6, \mathsf{P}_5, \boldsymbol{v}, \boldsymbol{x}, \mathsf{P}_4, \mathsf{P}_{2\_2}, \mathsf{P}_1, \mathsf{P}_3, \mathsf{P}_{1\_3}, \mathsf{P}_{1\_2}, \mathsf{P}_{1\_1}$, $\boldsymbol{w}, \mathsf{n}, H_3, H_2, H_1, \mathsf{A}, g_2, g_1, K, J, F$

Chapter 12

Related work and conclusions

In this last chapter I briefly review some of the other pieces of work that seem relevant to this thesis, and compare their approaches with my own. It should be understood that in some cases I am not comparing like with like since these other pieces of work had their own objectives and emphases; nevertheless, I hope that remarking on the differences is worthwhile. I also draw some conclusions about the success or otherwise of the case-study which in turn suggest some particular areas that further work might explore.

12.1 Related work

I did not make an attempt to be exhaustive in my review. Certain qualities made a piece of work seem more relevant to me; to start with, it should be large-scale, since formalising and presenting a larger development necessitates solving problems that do not occur in smaller "toy" projects. The involvement of either a type theory or an area of mathematics similar to that with which the case-study was concerned also made the project more important to me.

12.1.1 The GALOIS project

The case-study presented is designed to form one part of the larger formalisation of constructive Galois theory that comprises the GALOIS project. I review the other pieces of work on this project and their relation to the case-study.

An overall structure for GALOIS was suggested in a 1994 draft[Acz94] by Peter Aczel, which gives what is termed "a pre-formal blueprint for GALOIS"; an informal but careful presentation of the theory in the intended constructive style. A new independent paper[Dre95] by Andraes Dress suggested an alternative approach to some of the material, and the blueprint may be understood to have been tweaked to adopt these new ideas. The case-study is largely faithful to the resulting blueprint; the main change is to consider the extension fields L as subfields of a larger universal field F in order to fit better with the framework I implemented in type theory. Chapters 2 and 10 gave an explanation and justification of this alternative approach.

One of the aims of GALOIS was to encourage the collaboration of different authors in a large project and to examine the issues involved in making this feasible. The first work undertaken for GALOIS was an initial formalisation[Bar95] by Gilles Barthes of the unsolvability of the group of permutations on a set with five or more elements. This work is placed in context by a later report[Acz93]. My own MSc thesis[Bai94] formalised some GALOIS-related work on polynomial rings. However, both these formalisations would need to be almost entirely rewritten if they were to be used in combination with the case-study work, as they all feature different design decisions at a fairly low level. Although the act of having formalised some mathematics in one framework makes future formalisations much easier, there is still significant work involved in translating between the different frameworks.

The translation into a form compatible with my case-study would be significantly easier in the case of the final related development[Jon95], a formalisation of the decideability of dependence in finite vectorspaces that formed the MSc thesis of Alex Jones. This is largely because both this and the case-study were based on a common predicative library[Bai95]. (The library did precede the implementation of coercions, but it was modified to take advantage of these in a fairly straightforward way.)

12.1.2 Large-scale developments in LEGO

12.1.2.1 Pollack & McKinna's work on type theory

The largest LEGO development that has currently been undertaken is probably the ongoing formalisation of the theory of the λ-calculus and pure type systems (PTS) by James McKinna and Randy Pollack[McKP93, McKP97]. The authors were careful to choose concept representations that would be suitable for general use rather than for specific goals, and to structure their development through abstraction to make it a modular and extensible library. Currently it comprises more than a megabyte of source code.

It would have been pleasant to use a similar approach for my own case-study. Unfortunately, my own experience was that adding the necessary extra layer of abstraction pushed the time required for machine-checking beyond the bounds of feasibility. I found that the lesser computing resources to which I had access, and the disparate range of mathematical topics required for the case-study[1], meant that I had to tailor my own development toward the particular objective of proving the fundamental theorem, rather than building up a more modular and reuseable library of results.

[1] Pollack and McKinna note[McKP97] that type theory itself is "especially suitable for formalization because the objects are inductively constructed, their properties are proved by induction over structure, and there is little equality reasoning"

Although the presentation of their work to a reader has not been their fundamental concern, the authors experimented with a few different styles over the course of the papers[McKP93, Pol94, McKP97] they have written on their work. The original paper[McKP93] presented the source for parts of their formal proofs by copying it verbatim into the text. This sort of summary of verbatim source is probably the most common style found in the formalisations reviewed in this chapter. Since this manual extraction could introduce errors, for Pollack's PhD thesis[Pol94] the process was automated, with an external program extracting marked-up sections of the source documents. For the most recent report[McKP97], which needs to review a large amount of material in a small amount of space, the authors wrote an informal account that avoids the ASCII notation of LEGO, but which makes reference to formal names that can found in the source if the reader wishes to inspect the development at this level.

My own approach combines some aspects of the later two accounts and attempts to integrate the processing into the LEGO proof-checker itself. Source code is extracted and included in the thesis, but it is also pretty-printed, and the extraction and pretty-printing is performed by LEGO itself rather than by an external program. The case-study includes informal explanations, but these are interwoven more closely with formal terms of the type theory, again using the LEGO program itself to check the well-formedness of these terms.

Pretty-printing has both advantages and disadvantages. Although it is intended that the translation between the formal and pretty-printed material could be unambiguously reversed, this has not been formally checked and so the pretty-printing risks introducing parsing ambiguities into the presented material. Also, although moving the literate processes into the proof-checker has narrowed further the gap between the checked proof and the presented account, I have been able to choose which parts of the formal presentation to include and which to

suppress, so errors in my own judgement would still provide an opportunity for mistakes and misunderstandings.

12.1.2.2 Ruys' work on algebra

Mark Ruys worked on a formalisation of the fundamental theorem of algebra for his PhD thesis. Although the development was not completed, he proved a large body of results from algebra, including the binomial theorem and formulations of such notions as ordered fields and multivariate complex polynomials. His work is of interest to me for three particular reasons.

One is that it provides an example of where coercions would help to simplify expressions by allowing overloading of generic notions. Ruys uses a dense hierarchy of algebraic objects, and has to define new identifiers each time he wishes to lift concepts from a supertype to a subtype. Thus, for example, constructions defined on fields all have the suffix "`Fd`", and most of their definiens are explicitly inherited from earlier supertype objects such as rings through applying their own versions of the constructions to the ring underlying the field. Had implicit coercions been available to him the work would be both much shorter and easier to read.

Ruys implemented a framework for universal algebra with signatures and theories over those signatures defining models (which are the algebraic objects themselves.) This allows some more explicit inheritance of results and constructions (although again, in the presence of coercions it seems this process would have been easier.) I experimented with such a framework, but found it to be more trouble than it was worth for the purposes of the case-study, which does not crucially require any algebraic objects other than groups, fields and vectorspaces. Although duplicating some notions and proofs for the different objects was necessary, I found again that using concrete implementations of the structures (rather

than representing them through an abstraction layer) was necessary in order to keep the proof-checker running at a reasonable speed.

Ruys also implemented a mechanism to translate the LEGO source files into a hyperlinked collection of pages browseable through the World Wide Web. Even though the source has not been pretty-printed, the ability to browse a development at different depths of detail and to expand proofs or check definitions using hyperlinks adds a great deal to the ease of readability. It is my suspicion that such a style[Gru96] is probably a better method of presentation than the linear one employed by informal mathematics, certainly for libraries of formal results and quite possibly for broader categories of development. However, as noted in chapter 7, I chose not to complicate my own case-study further by adopting this paradigm as it is reasonably novel, especially to traditional mathematicians.

12.1.3 Developments in similar type theories

The proof-checker Coq[Coq96] and its type theory CIC (the Calculus of Inductive Constructions[Wer94]) are close cousins of LEGO and UTT. Many projects (especially proofs of program and protocol correctness) have been undertaken in Coq, although it doesn't currently have many facilities to aid large-scale developments that LEGO does not.

One large development of mathematics in Coq is a formalisation of constructive category theory[HS95] by Gérard Huet and Amokrane Saïbi. The paper cited is a summary of verbatim Coq source with informal commentary leading up to a proof of Yoneda's Lemma. Saïbi has built on this work further, proving further results and making use of Coq's new functionality for inheritance through coercion synthesis. A draft paper[Saï97b] presents this work in the same style as the previous one and the use of coercions makes the formal expressions noticeably easier to read.

12.1.4 Literate developments

As seen above, many large-scale formal developments have not addressed the issue of presenting the work to a reader at any level other than structuring their definitions and proofs. A completed formalisation is usually presented by quoting verbatim select parts of the formal source, and interspersing this with informal summary and commentary.

Some systems for formal mathematics take literateness and readability as a more important part of the formalisation. I review some of these in this section.

12.1.4.1 Deva and DevaWEB

Deva[Web93] is a formal system that was originally developed in the late 80's as a notation to support program development. It uses a descendant of the AUTOMATH family of type theories in which the types of λ-terms are themselves λ-terms. It has become a system used in the pursuit of literate formalisations, taking some direct inspiration from Knuth's literate programming[Knu92] in order to develop a DevaWEB environment[BRS93] which uses both the pretty-printing and also the reordering facilities found in the WEB paradigm of weaving and tangling.

The pretty-printing pays attention to the formatting of large typed expressions (propositions and proof objects for those propositions) in order that they can be read as conventional natural deduction proofs. For example, one novelty is use of the symbol "$\therefore$" instead of ":" in typing judgements so that a typed term $x : t$ can be read as "x, therefore t." Similarly some operators for the composition of proof-terms in either direction are used to indicate the flow of reasoning in a pretty-printed proof-term.

Some reasonably sized developments have been carried out in Deva, but it is still not clear that it has any more of the necessary support for organisation

of a very large-scale proof than other less literate systems such as LEGO. On the smaller scale it is more clearly successful, and has prompted some interesting investigation of formal proof styles[Sim96] using the Deva system and the DevaWEB environment as a basis.

12.1.4.2 Mizar and its journal

The Mizar project[Rud92] started with an attempt to formalise the language of mathematics (the so-called *mathematical vernacular*) in the early 70's. Since 1989 the major focus of the project has been to build up a database of mathematical results formally proven in the Mizar system.

The Mizar system, unlike most of those I consider, is based on classical set theory rather than type theory. One of the major objectives of the designers of the system was that it should be both useable and used by mathematicians, and they may be judged to have succeeded in this: their database of results is the largest in the world, containing tens of thousands of theorems.

The results proved within the system are published, and so making these results readable by human mathematicians and useable by other formal systems is an important concern in Mizar, working within the context of the wider QED project[QED94]. There is a printed journal[Miza], but probably the canonical source is the electronic version of this journal[Mizb], which presents abstracts together with the library of definitions and results that have been built up. These presentations are pretty-printed versions of the formal source, with the details of proofs suppressed but no additional informal commentary added. The resulting accounts have something in common with the uncommented pretty-printed formal development as presented in Appendix B, but with more flexible use of automatically produced natural language. Although somewhat dry in style, because the variety of styles of expression is limited as a result of the automation

process, the results are very readable.

12.1.4.3 The work of John Harrison

As mentioned in chapter 6, John Harrison has written[Har97] about an interesting distinction between two styles of formal proof. A *procedural* proof is one where one lists a sequence of actions to be performed to construct a proof object. The canonical example given of a procedural proof system is HOL; LEGO's refinement proofs are similar. However a *declarative* proof is one that proceeds by specifying the next result to be proved. In some cases the previous results that will be used to prove the new one may also be listed, but this is not strictly necessary as in some systems they may be found by a proof-search. One example of a declarative proof style is that found in the Mizar journals[Miza, Mizb]; my case-study proof style is also declarative for the most part, although it contains more details about the structure of the proof objects constructed.

Harrison perceives advantages and disadvantages of both styles in reading and writing nice formal proofs. He has also attempted to combine the best of both styles by implementing a "Mizar-mode for HOL"[Har96b]. In his paper[Har97] he considers the strengths and weaknesses of the two styles in terms of such qualities as writeability, readability, maintainability and efficiency of checking. I thought it would be worthwhile to review how the case-study system measures up in these regards. Proofs are as easy to write as they ever were in LEGO, although the writer tends to make use of the interactive features of the system for trying out ideas and then converting the procedural proofs constructed to declarative ones afterwards. Of course the whole point of the proof style is that I aim to produce readable proofs. My proofs are not very modular, and this makes them difficult to maintain; I have often has to redo later work as a result of changing some earlier choices. Efficiency of checking is also not as good as it could be, and sometimes

the need to ensure that checking remained practical meant I had to make choices that were non-optimal in other regards. A new proof system in which some of the theoretically useful features (*e.g.* more abstraction through inductive types and cutting) were implemented more efficiently would certainly be desirable.

12.1.5 Computer algebra systems

As well as checking formal proofs, machine assistants are also used in algebraic computation. AXIOM[Dav] is a good example of such a computer algebra system. The machine plays the role of a *symbolic solver* and the user communicates with it in terms of strongly-typed formulae and specifications. The algebraic subject matter makes such systems interesting in relation to my work, as does the need for them to present large bodies of results. (AXIOM uses a hypertext-based interface for this.)

Most computer algebra systems are just computational tools and they ignore the proof properties of the systems with which they deal. However, there is interest from the world of computer algebra systems in beginning to cover their formal properties as well. In the other direction, the proof-checking community is beginning to become interested in the high-speed computational and rewriting abilities of these systems. One can expect there to be useful combinations of these systems in the near future.

Some work has already been performed along these lines. Relatively slow proof-checkers have been persuaded to farm out their rewriting and symbolic search work to faster external computer algebra systems. Also Paul Jackson[Jac95] has incorporated some features from computer algebra systems into the foundations of the NuPrl[NuP86] proof system.

12.1.6 The formalisation process

I shall briefly review some papers relevant to formalisation in general and to literate formalisation in particular.

Polemics to motivate the formalisation of mathematics can be found in the QED manifesto[QED94]. I have already mentioned that John Harrison[Har97] and Martin Simons[Sim96] have written papers on the style of formal proofs, as has Leslie Lamport[Lam94]. Harrison has also written a history of formalisation efforts[Har96a], including a discussion of recent attempts using computers that make the formalist program a pragmatic possibility, rather than one of only theoretical interest.

To successfully formalise mathematics in a style that present-day mathematicians will be willing to read, one needs a formal language that faithfully represents the style of language found in informal mathematics, the so-called *mathematical vernacular*. There have been several pieces of work with this phrase as a title over the past few decades. Forming a formal yet flexible language has been attempted by *e.g.* the Mizar group[Rud92] in the 70's, de Bruijn[deB94] in the 80's, and now by a group at Durham[CL97].

Machine-checked formalisations of a result are good evidence for it being true, but one needs to be confident that one has received a correctly checked formalisation and that one correctly understands what result has been proved, in order to "believe" a machine-checked proof. These issues are examined in the context of modern computer proof-assistants in a paper by Randy Pollack[Pol97] that I have cited several times already in this thesis.

I quickly review here what would need to be done to make the case-study formalisation "believeable" in Pollack's sense. To independently check the formalisation should not be too much of a problem because all syntactical short-cuts can be unambiguously expanded at the source code level. This suffices to check

the formalisation itself. At the level of the pretty-printed account in the appendix, I believe the presentation can be parsed automatically and unambiguously at least in principle and so one could understand fully the result that has been proven by reading that account, although the material is not actually presented in a form that would currently make that very easy.

12.2 Conclusions

In this final section I review the ways in which the case-study development and the methodologies behind it met, or did not meet, my objectives for the work. I also extrapolate the work done a little, judging which parts seemed to be going in the right direction and which would be better rethought. In turn this suggests some avenues for literate large-scale formalisations to explore in the future.

12.2.1 Areas of success

At the most basic level, the case-study formalisation was successful in that the results required were formulated and proven. One major subresult was left unproved, but this was because other equally time-consuming matters were decided to constitute more valuable work rather than because it would have been any more difficult to prove. I experimented with various working methodologies and fundamental frameworks. The one on which I settled seems effective and flexible; it allowed a faithful formulation of the required material, and the proof style resulted in the development having a more readable shape than most formalisations.

On the level of improving the readability of expressions in the type theory, the implemented mechanisms for coercion synthesis formed a very good bridge between formal rigour and the natural informal idioms. Coercion synthesis is seen

to enrich the expressiveness of a type theory a great deal, without introducing ambiguities or other drawbacks. I would count the material on coercions as the most immediately useful and worthwhile work that I did, and see no reason not to include them in every modern proof-checker based on a type theory.

The tools and mechanisms implemented for literate formalisation and pretty-printing were largely successful and the combined features constituted an original methodology and literate environment. The ability to pretty-print from within the proof-checker and thus to typecheck and "quote" formal terms in informal commentary seems especially valuable. The resulting environment can be used to present proofs at various levels of detail.

12.2.2 Weaknesses in the work

The eventual aim of literate formalisation must be to provide tools through which a mathematician may formalise a proof by translating it into a format which is parsed and processed to produce a document which is as easy to read as the best informal developments published in journals today, but is also essentially unambiguous and that can be checked by a machine. Of course as we take the first steps towards this aim, the work currently falls far short of this objective.

Although the case-study pieces presented are far easier to read than formal source code, they are still clumsy and tedious compared to the fluent and idiomatic language that may be used within informal mathematics. An experienced mathematician will find chapter 10 much harder to read than a carefully written informal account. This is partly because of my own inexperience with writing readable mathematics, but also the literate environment itself still has many problems.

There are also some problems with the formalisation framework. It is still necessary to distinguish between objects and subobjects to a greater degree than is

normal in conventional mathematics. Also some definitions were skewed towards allowing this particular development to proof-check in a reasonable amount of time rather than implementing them in a modular and reuseable way.

There still remains work to do on coercions, especially in implementing them in a more efficient manner. Some other possible extensions were discussed in chapter 5.

The pretty-printing system works, but it is does not yet provide the ability to write formal mathematics using all of the informal tricks and conventions which mathematicians are familiar. These include some simple lexical features, and also some semantic notions such as proper overloading. The coherency of the overloading used in the case-study is supported by the formal environment, but it is not guaranteed. The same can be said for the agreement of the formal proofs with the informal commentary and summaries.

A final note is that even a formal proof cannot be an absolute guarantee of truth. In his PhD thesis[Pol94], Randy Pollack writes

> A minor theme of this thesis is that there is no such thing as absolute certainty, and machine verification of various kinds does not alter that common truth about the world.

This is a theme with which I identify. However I do hope that I have taken some new and worthwhile steps in the direction of this impossible goal, allowing a reader to be more confident in trusting the case-study than they would normally be with such a formalisation.

12.2.3 Suggestions for future directions

The above two sections suggest many directions that might be pursued in further work, either to develop the good aspects or to rethink the bad ones. I emphasise three in particular.

Although coercions are powerful tools, their potential has not yet been fully realised. Alternative approaches to coherency and coercion generation were discussed in chapter 5. The most useful extension would probably be the implementation of a safe form of overloading through the use of coercions from unit types.

Mathematicians could consider reassessing how they present their accounts to take advantage of the searching abilities provided by structured databases and browsing facilities provided by hypertext. This functionality is especially important in formal mathematics where one may wish to chase definitions, or to review a development at various levels of detail, anywhere from the precise intricacies to a very general overview.

In order to make good progress on the modularity of large-scale proofs and their literate presentation, I perceive a need for the next generation of proof-checkers to be built with support for these aspects of formalisation in mind, with information hiding, user-specified grammars for both input and output, and the ability to take a less linear direction in proofs.

Bibliography

[Acz93] P. Aczel. *Galois: a theory development project*. Representation of Mathematics in Logical Frameworks, Turin; 1993.

[Acz94] P. Aczel. *Notes towards a formalisation of Constructive Galois Theory*. Draft report; 1994
<ftp://ftp.cs.man.ac.uk/pub/applied_logic/PeterAczel/galois-theory.ps.gz>.

[Bai94] A. Bailey. *Representing algebra in LEGO*. MSc thesis, Department of Computer Science, University of Edinburgh, 1994.

[Bai95] A. Bailey. *Predicative logic, setoid and mappings as a framework for universal algebra in LEGO*. Draft, included as an appendix to [Jon95]; 1995.

[Bai98] A. Bailey. *Coercion synthesis in computer implementations of type-theoretic frameworks*. Submitted to Proceedings of TYPES'97, Springer-Verlag; expected to be published 1998.

[Coq96] B. Barras, S. Boutin, C. Cornes, J. Courant, J. Fillâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, B. Werner. *The Coq proof assistant reference manual*. INRIA-Rocquencourt and CNRS-ENS Lyon; 1996.

[Bar95] G. Barthe. *Formalising algebra in type theory: fundamentals and applications to group theory.* Faculty of Mathematics and Informatics, University of Nijmegen; 1995.

[Bar96] G. Barthe. *Implicit coercions in type systems.* Proceedings of TYPES'95, Springer-Verlag; 1996.

[BT95] G. Betarte, A. Tasistro. *Extension of Martin-Löf 's theory of types with record types and subtyping.* Two manuscripts; 1995.

[BRS93] M. Biersack, R. Raschke, M. Simons. *The DevaWEB system: Introduction, tutorial, user manual, and implementation.* Technische Universität Berlin report 93–39; 1993.

[BCGS91] V. Breazu-Tannen, T. Coquand, C. Gunter, A. Scedrov. *Inheritance as Implicit Coercion.* Information and Computation 93, July 1991.

[deB94] N. de Bruijn. *The mathematical vernacular, a language for mathematics with typed sets.* 1994.

[BMcK93] R. Burstall, J. McKinna. *Deliverables: a categorical approach to program development in type theory.* Proceedings of Mathematical Foundations for Computer Science, MFCS '93, Gdansk; 1993.

[CL97] P. Callaghan, Z. Luo. *Linguistic categories in mathematical vernacular and their type-theoretic semantics* Extended abstract in Proceedings of Logical Aspects of Computational Linguistics; 1997.

[NuP86] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, S. Smith. *Implementing mathematics with the NuPrl development system.* Prentice-Hall, New Jersey; 1986

[CH88] Th. Coquand, G. Huet. *The calculus of constructions*. Information and Computation 76; 1988.

[Dav] J. Davenport. *The AXIOM system*. NAG technical report TR52 ATR/3 NP2492.
<http://extweb.nag.co.uk/doc/TechRep/PS/atr3.ps.Z>.

[Dre95] A. Dress. *One more shortcut to Galois Theory*. Advances in Mathematics 110; 1995.

[Gol81] W. Goldfarb. *The undecidability of the second-order unification problem*. Theoretical Computer Science 13; 1981.

[Gru96] J. Grundy. *A browsable format for proof presentation*. Proceedings of the Finnish Artificial Intelligence Society Symposium: Logic, Mathematics and the Computer; 1996.

[Har96a] J. Harrison. *Formalized Mathematics*. Technical report 36, Turku Centre for Computer Science; 1996.
<http://www.cl.cam.ac.uk/users/jrh/papers/form-math3.html>.

[Har96b] J. Harrison. *A Mizar mode for HOL*. Proceedings of Theorem Proving in Higher Order Logics, 9th International Conference, LNCS volume 1125, Springer-Verlag; 1996.
<http://www.cl.cam.ac.uk/users/jrh/papers/mizar.html>.

[Har97] J. Harrison. *Proof style*. Technical report 410, University of Cambridge Computer Laboratory; 1997.
<http://www.cl.cam.ac.uk/users/jrh/papers/style.html>.

[HS95] G. Huet, A. Saïbi. *Constructive category theory*. Proceedings of the joint CLICS-TYPES Workshop on Categories and Type Theory, Goteborg; 1995.

[Jac95] P. Jackson. *Enhancing the Nuprl proof development system and applying it to computational abstract algebra*. PhD thesis, Cornell University; 1995.

[Jon95] A. Jones. *The formalization of linear algebra in LEGO: the decidable dependency theorem*. Master's thesis, Department of Mathematics, University of Manchester, 1995.

[Jon96] C. Jones, S. Maharaj. The LEGO library; 1996. `<http://www.dcs.ed.ac.uk/ lego/html/library/newlib.html>`.

[Knu92] D. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27; 1992. ISBN 0-937073-80-6.

[Lam94] L. Lamport. *How to write a proof*. Research report 94, DEC Systems Research Center; 1993.

[Luo90] Z. Luo. *An extended calculus of constructions*. PhD thesis, Department of Computer Science, University of Edinburgh, 1990.

[Luo94] Z. Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press; 1994.

[Luo97] Z. Luo. *Coercive subtyping in type theory*. Proceedings of Annual Conference of the European Association for Computer Science Logic, Utrecht, LNCS volume 1258, Springer-Verlag; 1997. A later version of this paper, *Coercive subtyping*, has also been submitted for publication.

[LP92] Z. Luo, R. Pollack. *LEGO proof development system: user's manual*. Technical report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh; 1992.

Readers should also consult the associated *Incremental changes in LEGO* documents by C. Jones and R. Pollack; 1993, 1994.

[McKP93] J. McKinna, R. Pollack. *Pure Type Systems formalized*. Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, LNCS volume 664, Springer-Verlag; 1993. <ftp://ftp.dcs.ed.ac.uk/pub/lego/formalPTS.ps.Z>.

[McKP97] J. McKinna, R. Pollack. *Some lambda calculus and type theory formalized*. Submitted to Journal of Automated Reasoning; 1997. <ftp://ftp.dcs.ed.ac.uk/pub/lego/McKinnaPollack97.ps.gz>.

[MLof84] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis; 1984.

[Miza] R. Matuszewski (Editor.) *Formalized Mathematics, Volumes 1, 2 and 3*. Universite Catholique de Louvain - Fondation Philippe le Hodey, Brussels; 1990, 1991 and 1992. (There also exists an electronic form of this journal[Mizb].)

[Mizb] Various authors involved in the Mizar project. *Journal of formalized mathematics, Volumes 1–9*. 1989–1997. (There also exists a print form of this journal[Miza].) <http://mizar.uw.bialystok.pl/JFM/>.

[Nip90] T. Nipkow. *Higher-order unification, polymorphism, and subsorts*. Proceedings of the Second International Workshop on Conditional and Typed Rewriting Systems, LNCS volume 516, Springer-Verlag; 1990.

[NPS90] B. Nordström, K. Petersson, J. Smith. *Programming in Martin-Löf's type theory*. Oxford University Press, 1990.

[Pau93] L. Paulson. *The Isabelle reference manual*. The University of Cambridge Computer Laboratory technical report 283; 1993.

[Pol90] R. Pollack. *Implicit syntax*. Informal Proceedings of the First Workshop on Logical Frameworks (Antibes); 1990.

[Pol94] R. Pollack. *The theory of LEGO: a proof-checker for the Extended Calculus of Constructions*. PhD thesis, Department of Computer Science, University of Edinburgh, 1994.
<ftp://ftp.dcs.ed.ac.uk/pub/lego/thesis-pollack.ps.gz>.

[Pol97] R. Pollack. *How to believe a machine-checked proof*. submitted to Twenty-Five Years of Constructive Type Theory: Proceedings of the Venice Meeting.
<ftp://ftp.dcs.ed.ac.uk/pub/lego/pollack-belief.ps.gz>.

[QED94] Anonymous. *The QED manifesto*. 12th International Conference on Automated Deduction, Lecture Notes in Computer Science volume 814, Springer-Verlag; 1994.
<http://www.mcs.anl.gov/qed/>.

[Rud92] P. Rudnicki. *An overview of the Mizar project*. Proceedings of the 1992 Workshop on Types for Proofs and Programs, Chalmers University of Technology; 1992.

[Saï97a] A. Saïbi. *Typing algorithm in type theory with inheritance*. 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris; 1997.

[Saï97b] A. Saïbi. *Théorie constructive des catégories*. Draft; 1997.
<http://pauillac.inria.fr/ saibi/Cat_monographie.ps>.

[Sim96] M. Simons. *The presentation of formal proofs*. Technischen Universität Berlin thesis; 1996.

[Ste79] I. Stewart. *Galois theory*. Chapman and Hall, second edition; 1979. ISBN 0-412-34550-1.

[Web93] M. Weber. *Definition and basic properties of the Deva meta-calculus*. Formal Aspects of Computing, volume 5; 1993.

[Wen97] M. Wenzel. *Type classes and overloading in higher-order logic*. Theorem Proving in Higher Order Logics, LNCS volume 1275, Springer-Verlag; 1997.

[Wer94] Calculus of Inductive Constructions B. Werner. *Une théorie des constructions inductives*. Doctoral thesis, Université Paris 7; 1994.

Appendix A

Resources

Source code, executables, documentation and other resources for the LEGO proof-checker are available electronically. The home page for LEGO at Edinburgh is:

<http://www.dcs.ed.ac.uk/home/lego/>

The next release of the proof-checker will probably be accompanied by a beta-test version that features coercion synthesis. This will be based on my code and therefore should be compatible with my use of coercions in this thesis.

The extended version of LEGO that I used for the case-study development, and the source code for the checked development, should also be available electronically, here at Manchester. Since I am not certain what will happen to my FTP space following the end of my studies here, and I am not planning to continue in academia, I shall provide a single URL that I hope can remain available, and from where I can link dynamically to wherever the material ends up being archived.

<http://www.cs.man.ac.uk/~baileya/thesis.html>

This page intentionally left blank (title page for Volume II)

This page intentionally left blank (Index for Volume II, first page)

This page intentionally left blank (Index for Volume II, second page)

Appendix B

Formalisation

This is the full pretty-printed case-study development, in the order that it is checked by LEGO. It is presented mostly for reference purposes: a reader can use it to chase back definitions and to read the full details of any proof. However, it is largely uncommented, and the presentation has not been tweaked for maximum readability. Ideally I would have liked to have had time to tidy it further so as to make it easier to consult and read small sections of, as in its current form it does not represent the full potential of the current literate environment. Even so, I believe that reading this sectioned and pretty-printed version of the development is still far easier than reading the ASCII source files would be.

The first part of the development (pages 255–368) sets up the mathematical framework in which the case-study itself will take place. Its most important parts are summarised in the corresponding chapters 8 and 9. A small portion of this code (starting on page 364) is the uncommented version of the development segment presented more fully in chapter 11.

The second part of the development (pages 368–413) is the detailed and uncommented version of the case-study proof of the fundamental theorem of Galois theory itself. This is the part of the proof that is presented at a higher level of abstraction (omitting many of the lower level details presented here) in chapter 10.

B.1 Logic and basic types

B.1.1 The universe

B.1.1.1 The universe of propositions

Most propositions are types in the lowest universe; occasionally we have to construct a type that we would consider to be a proposition but that lies in the next universe up.

▷ Define $\mathsf{prop} = \mathsf{Type}_0 : \mathsf{Type}_1$

▷ Define $\mathsf{prop}_1 = \mathsf{Type}_1 : \mathsf{Type}_2$

▷ Unless otherwise specified, by default $p, p_1, p_2, p_3 : \mathsf{prop}$

B.1.2 Conjunction

B.1.2.1 Logical conjunction

▷ Allow $\wedge$ to be written infix

▷ Define $\wedge = [\lambda p_1, p_2]\ p_1\ \#\ p_2 : \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}$

▷ Introduce $p_1, p_2 \mid \mathsf{prop}$ and suppose $H : p_1 \wedge p_2$

▷ Define frozen $\mathsf{pair} = [\lambda P_1 : p_1]\ [\lambda P_2 : p_2]\ (P_1, P_2) : p_1 \to p_2 \to p_1 \wedge p_2$

▷ Define frozen $\mathsf{fst} = H._{\mathbf{1}} : p_1$

▷ Define frozen $\mathsf{snd} = H._{\mathbf{2}} : p_2$

▷ Discharge H, p_2, p_1

▷ Define $\bigwedge_3 = [\lambda p_1, p_2, p_3]\ (p_1 \wedge p_2) \wedge p_3 : \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}$

▷ Introduce $p_1, p_2, p_3 \mid \mathsf{prop}$ and suppose $H : \bigwedge_3\ p_1\ p_2\ p_3$

▷ Define frozen $\mathsf{pair}_3 = [\lambda P_1 : p_1]\ [\lambda P_2 : p_2]\ [\lambda P_3 : p_3]\ \mathsf{pair}\ (\mathsf{pair}\ P_1\ P_2)\ P_3$
$: p_1 \to p_2 \to p_3 \to \bigwedge_3\ p_1\ p_2\ p_3$

▷ Define frozen $\mathsf{fst}_3 = H.\mathsf{fst}.\mathsf{fst} : p_1$

▷ Define frozen $\mathsf{snd}_3 = H.\mathsf{fst}.\mathsf{snd} : p_2$

▷ Define frozen $\mathsf{thd}_3 = H.\mathsf{snd} : p_3$

▷ Discharge H, p_3, p_2, p_1

▷ Explicitly overload the identifier $\wedge$

▷ Define $\wedge = [\lambda p_1, p_2 : \mathsf{prop}_1]\ p_1\ \#\ p_2 : \mathsf{prop}_1 \to \mathsf{prop}_1 \to \mathsf{prop}_1$

▷ Introduce $p_1, p_2 \mid \mathsf{prop}_1$ and suppose $H : p_1 \wedge p_2$

▷ Define frozen $\mathsf{pair'} = [\lambda P_1 : p_1]\ [\lambda P_2 : p_2]\ (P_1, P_2) : p_1 \to p_2 \to p_1 \wedge p_2$

▷ Define frozen $\mathsf{fst'} = H._{\mathbf{1}} : p_1$

▷ Define frozen $\mathsf{snd'} = H._{\mathbf{2}} : p_2$

▷ Discharge H, p_2, p_1

▷ Explicitly overload the identifier $\bigwedge_3$

▷ Define $\bigwedge_3 = [\lambda p_1, p_2, p_3 : \mathsf{prop}_1]\ (p_1 \wedge p_2) \wedge p_3$

$: \mathsf{prop}_1 \to \mathsf{prop}_1 \to \mathsf{prop}_1 \to \mathsf{prop}_1$

▷ Introduce $p_1, p_2, p_3 \mid \mathsf{prop}_1$ and suppose $H : \bigwedge_3\ p_1\ p_2\ p_3$

▷ Define frozen $\mathsf{pair}_3\mathsf{'} = [\lambda P_1 : p_1]\ [\lambda P_2 : p_2]\ [\lambda P_3 : p_3]\ \mathsf{pair'}\ (\mathsf{pair'}\ P_1\ P_2)\ P_3$

$: p_1 \to p_2 \to p_3 \to \bigwedge_3\ p_1\ p_2\ p_3$

▷ Define frozen $\mathsf{fst}_3\mathsf{'} = H.\mathsf{fst'}.\mathsf{fst'} : p_1$

▷ Define frozen $\mathsf{snd}_3\mathsf{'} = H.\mathsf{fst'}.\mathsf{snd'} : p_2$

▷ Define frozen $\mathsf{thd}_3\mathsf{'} = H.\mathsf{snd'} : p_3$

▷ Discharge H, p_3, p_2, p_1

B.1.2.2 Logical equivalence

▷ Allow $\leftrightarrow$ to be written infix

▷ Explicitly overload the identifier $\leftrightarrow$

▷ Define $\leftrightarrow = [\lambda p_1, p_2]\ (p_1 \to p_2) \wedge (p_2 \to p_1) : \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}$

▷ Define $\leftrightarrow = [\lambda p_1, p_2 : \mathsf{prop}_1]\ (p_1 \to p_2) \wedge (p_2 \to p_1) : \mathsf{prop}_1 \to \mathsf{prop}_1 \to \mathsf{prop}_1$

B.1.2.3 Results concerning logical equivalence

▷ Prove iff_refl : $\{\forall p\}\ p \leftrightarrow p$
$= [\lambda p]\ \mathsf{pair}\ ([\lambda P : p]\ P)\ ([\lambda P : p]\ P)$

▷ Prove iff_symm : $\{\forall p_1, p_2 \mid \mathsf{prop}\}\ (p_1 \leftrightarrow p_2) \to p_2 \leftrightarrow p_1$
$= [\lambda p_1, p_2 \mid \mathsf{prop}]\ [\lambda Q : p_1 \leftrightarrow p_2]\ \mathsf{pair}\ Q.\mathsf{snd}\ Q.\mathsf{fst}$

▷ Prove iff_tran : $\{\forall p_1, p_2, p_3 \mid \mathsf{prop}\}\ (p_1 \leftrightarrow p_2) \to (p_2 \leftrightarrow p_3) \to p_1 \leftrightarrow p_3$
$= [\lambda p_1, p_2, p_3 \mid \mathsf{prop}]\ [\lambda Q_1 : p_1 \leftrightarrow p_2]\ [\lambda Q_2 : p_2 \leftrightarrow p_3]$
$\mathsf{pair}\ ([\lambda P : p_1]\ Q_2.\mathsf{fst}\ (Q_1.\mathsf{fst}\ P))\ ([\lambda P : p_3]\ Q_1.\mathsf{snd}\ (Q_2.\mathsf{snd}\ P))$

B.1.3 Disjunction

B.1.3.1 The sum of two types

▷ Allow $\oplus$ to be written infix

▷ Introduce t_1, t_2 : Type_0; globally declare $\oplus$: Type_0; left : $t_1 \to \oplus$; right : $t_2 \to \oplus$ and disjoint_sum_elim : $\{\Pi C\_disjoint\_sum : \oplus \to \mathsf{Type}_1\}$
$(\{\Pi x_2 : t_1\}\ C\_disjoint\_sum\ (\mathsf{left}\ x_2)) \to$
$(\{\Pi x_1 : t_2\}\ C\_disjoint\_sum\ (\mathsf{right}\ x_1)) \to \{\Pi z : \oplus\}\ C\_disjoint\_sum\ z$

▷ Reductions: $[\lambda C\_disjoint\_sum : \oplus \to \mathsf{Type}_1]\ [\lambda f\_left : \{\Pi x_2 : t_1\}$
$C\_disjoint\_sum\ (\mathsf{left}\ x_2)]\ [\lambda f\_right : \{\Pi x_1 : t_2\}\ C\_disjoint\_sum\ (\mathsf{right}\ x_1)]$
$[\lambda x_2 : t_1]\ [\lambda x_1 : t_2]$
disjoint_sum_elim $C\_disjoint\_sum\ f\_left\ f\_right\ (\mathsf{left}\ x_2) \Longrightarrow f\_left\ x_2$
$\|$ disjoint_sum_elim $C\_disjoint\_sum\ f\_left\ f\_right\ (\mathsf{right}\ x_1) \Longrightarrow$
$f\_right\ x_1$

▷ Discharge t_2, t_1

▷ Introduce t_1, t_2 | Type_0

▷ Define disjoint_sum_rec $= [\lambda t \mid \mathsf{Type}_1]$ disjoint_sum_elim $t_1\ t_2\ ([\lambda\_ : t_1 \oplus t_2]\ t)$
: $\{\Pi t \mid \mathsf{Type}_1\}\ (t_1 \to t) \to (t_2 \to t) \to (t_1 \oplus t_2) \to t$

▷ Discharge t_2, t_1

B.1.3.2 Logical disjunction

▷ Allow $\vee$ to be written infix

▷ Define $\vee = [\lambda p_1, p_2]\ p_1 \oplus p_2 : \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}$

▷ Introduce $p_1, p_2 \mid \mathsf{prop}$; suppose $P_1 : p_1$ and $P_2 : p_2$

▷ Define $\mathsf{inl} = \mathsf{left}\ p_1\ p_2\ P_1 : p_1 \vee p_2$

▷ Define $\mathsf{inr} = \mathsf{right}\ p_1\ p_2\ P_2 : p_1 \vee p_2$

▷ Discharge P_2, P_1

▷ Define $\mathsf{case} = [\lambda t \mid \mathsf{Type}_1]\ [\lambda H : p_1 \vee p_2]\ [\lambda T_1 : p_1 \to t]\ [\lambda T_2 : p_2 \to t]$

$\mathsf{disjoint\_sum\_rec}\ T_1\ T_2\ H$

$: \{\Pi t \mid \mathsf{Type}_1\}\ (p_1 \vee p_2) \to (p_1 \to t) \to (p_2 \to t) \to t$

▷ Define $\mathsf{case\_elim} = [\lambda t : (p_1 \vee p_2) \to \mathsf{Type}_1]\ [\lambda H : p_1 \vee p_2]\ [\lambda T_1 : \{\Pi x : p_1\}$

$t\ (\mathsf{inl}\ x)]\ [\lambda T_2 : \{\Pi x : p_2\}\ t\ (\mathsf{inr}\ x)]\ \mathsf{disjoint\_sum\_elim}\ p_1\ p_2\ t\ T_1\ T_2\ H$

$: \{\Pi t : (p_1 \vee p_2) \to \mathsf{Type}_1\}\ \{\Pi H : p_1 \vee p_2\}\ (\{\Pi x : p_1\}\ t\ (\mathsf{inl}\ x)) \to$

$(\{\Pi x : p_2\}\ t\ (\mathsf{inr}\ x)) \to t\ H$

▷ Discharge p_2, p_1

▷ Prove $\mathsf{inl\_is\_true} : \{\Pi p_1 \mid \mathsf{prop}\}\ p_1 \to \{\Pi p_2\}\ p_1 \vee p_2$
$= [\lambda p_1 \mid \mathsf{prop}]\ [\lambda P_1 : p_1]\ [\lambda p_2]\ \mathsf{inl}|p_1|p_2\ P_1$

▷ Prove $\mathsf{inr\_is\_true} : \{\Pi p_1\}\ \{\Pi p_2 \mid \mathsf{prop}\}\ p_2 \to p_1 \vee p_2$
$= [\lambda p_1]\ [\lambda p_2 \mid \mathsf{prop}]\ [\lambda P_2 : p_2]\ \mathsf{inr}|p_1|p_2\ P_2$

▷ Define $\bigvee_3 = [\lambda p_1, p_2, p_3]\ p_1 \vee (p_2 \vee p_3) : \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}$

▷ Introduce $p_1, p_2, p_3 \mid \mathsf{prop}$; suppose $P_1 : p_1$; $P_2 : p_2$ and $P_3 : p_3$

▷ Define $\mathsf{inl}_3 = \mathsf{inl}|p_1|(p_2 \vee p_3)\ P_1 : \bigvee_3\ p_1\ p_2\ p_3$

▷ Define $\mathsf{inm}_3 = \mathsf{inr}|p_1\ (\mathsf{inl}|p_2|p_3\ P_2) : \bigvee_3\ p_1\ p_2\ p_3$

▷ Define $\mathsf{inr}_3 = \mathsf{inr}|p_1\ (\mathsf{inr}|p_2|p_3\ P_3) : \bigvee_3\ p_1\ p_2\ p_3$

▷ Define $\mathsf{case}_3 = [\lambda t \mid \mathsf{Type}_1]\ [\lambda H : \bigvee_3\ p_1\ p_2\ p_3]\ [\lambda T_1 : p_1 \to t]\ [\lambda T_2 : p_2 \to t]$

$[\lambda T_3 : p_3 \to t]\ \mathsf{case}\ H\ T_1\ ([\lambda H' : p_2 \vee p_3]\ \mathsf{case}\ H'\ T_2\ T_3)$

$: \{\Pi t \mid \mathsf{Type}_1\}\ (\bigvee_3\ p_1\ p_2\ p_3) \to (p_1 \to t) \to (p_2 \to t) \to (p_3 \to t) \to t$

▷ Discharge $P_3, P_2, P_1, p_3, p_2, p_1$

B.1.4 Truth and falsehood

B.1.4.1 The unit type

▷ Globally declare $\top$: Type_0; $\star$: $\top$ and unit_elim

: $\{\Pi C\_unit : \top \to \mathsf{Type}_1\}\ (C\_unit\ \star) \to \{\Pi z : \top\}\ C\_unit\ z$

▷ Reductions: $[\lambda C\_unit : \top \to \mathsf{Type}_1]\ [\lambda f\_star : C\_unit\ \star]$

$\mathsf{unit\_elim}\ C\_unit\ f\_star\ \star \Longrightarrow f\_star$

▷ Define $\mathsf{unit\_rec} = [\lambda t \,|\, \mathsf{Type}_1]\ \mathsf{unit\_elim}\ ([\lambda\_ : \top]\ t)$: $\{\Pi t \,|\, \mathsf{Type}_1\}\ t \to \top \to t$

B.1.4.2 Logical truth

▷ Define $\mathsf{true} = \top$: prop

▷ Define frozen $\tau = \star$: true

B.1.4.3 The empty type

▷ Globally declare $\bot$: Type_0 and empty_elim

: $\{\Pi C\_empty : \bot \to \mathsf{Type}_1\}\ \{\Pi z : \bot\}\ C\_empty\ z$

▷ Define frozen $\mathsf{empty\_rec} = [\lambda t : \mathsf{Type}_1]\ \mathsf{empty\_elim}\ ([\lambda\_ : \bot]\ t)$

: $\{\Pi t : \mathsf{Type}_1\}\ \bot \to t$

B.1.4.4 Logical falsehood

▷ Define $\mathsf{false} = \bot$: prop

▷ Prove ex_falso : $\{\forall p\}\ \mathsf{false} \to p$
$= [\lambda p]\ \mathsf{empty\_rec}\ p$

B.1.4.5 Logical negation

▷ Allow $\neg$ to be written prefix

▷ Define $\neg = [\lambda p]\ p \to \mathsf{false} : \mathsf{prop} \to \mathsf{prop}$

B.2 Sets

B.2.1 Basic definitions

B.2.1.1 Relations

▷ Allow $\sim$ to be written infix

▷ Allow $\sim'$ to be written infix

▷ Define $\mathsf{rel} = [\lambda t : \mathsf{Type}_0]\ t \to t \to \mathsf{prop} : \mathsf{Type}_0 \to \mathsf{Type}_1$

▷ Introduce $t \mid \mathsf{Type}_0$ and $\sim, \sim' : \mathsf{rel}\ t$

▷ Define $\mathsf{is\_refl} = \{\forall x : t\}\ x \sim x : \mathsf{prop}$

▷ Define $\mathsf{is\_symm} = \{\forall x, y \mid t\}\ (x \sim y) \to y \sim x : \mathsf{prop}$

▷ Define $\mathsf{is\_tran} = \{\forall x, y, z \mid t\}\ (x \sim y) \to (y \sim z) \to x \sim z : \mathsf{prop}$

▷ Define $\mathsf{is\_subrelation} = \{\forall x, y \mid t\}\ (x \sim y) \to x \sim' y : \mathsf{prop}$

▷ Discharge but keep $\sim', \sim$

▷ Define $\mathsf{is\_same\_relation} = (\sim.\mathsf{is\_subrelation}\ \sim') \wedge (\sim'.\mathsf{is\_subrelation}\ \sim) : \mathsf{prop}$

▷ Discharge $\sim', \sim, t$

B.2.1.2 Sets

A set is a type of elements $el : \mathsf{Type}_0$ together with an equivalence relation defined upon it.

▷ Define $\mathsf{set\_axioms} = [\lambda el \mid \mathsf{Type}_0]\ [\lambda eq : \mathsf{rel}\ el]$

$\bigwedge_3 eq.\mathsf{is\_refl}\ eq.\mathsf{is\_symm}\ eq.\mathsf{is\_tran} : \{\Pi el \mid \mathsf{Type}_0\}\ (\mathsf{rel}\ el) \to \mathsf{prop}$

▷ Define $\mathsf{set} = \langle \Sigma el : \mathsf{Type}_0 \rangle\ \langle \Sigma eq : \mathsf{rel}\ el \rangle\ \mathsf{set\_axioms}\ eq : \mathsf{Type}_1$

▷ Unless otherwise specified, by default $S, T, S_0, S_1, S_2, S_3, S_4 \mid \mathsf{set}$

▷ Define kind-coercion $\mathsf{el} = [\lambda S : \mathsf{set}]\ S_{.\mathbf{1}} : \mathsf{set} \to \mathsf{Type}_0$

▷ Define $\mathsf{equal\_in} = [\lambda S : \mathsf{set}]\ S_{.\mathbf{2}\cdot\mathbf{1}} : \{\Pi S : \mathsf{set}\}\ \mathsf{rel}\ S$

▷ Introduce S

▷ Allow $=$ to be written infix

For the sake of readability, I write equality in a set as a standard equals sign, "=". The use of this symbol *within* expressions should not be confusing, as the symbols for definitional equality and convertibility that are used to relate expressions are written slightly differently, as "$\mathbin{=\!\!=}$" and "$\cong$" respectively.

▷ Define $= = S_{.\mathbf{2}\cdot\mathbf{1}}$: rel S

▷ Prove refl : =.is_refl
$= S_{.\mathbf{2}\cdot\mathbf{2}}.\mathsf{fst}_3$

▷ Prove symm : =.is_symm
$= S_{.\mathbf{2}\cdot\mathbf{2}}.\mathsf{snd}_3$

▷ Prove tran : =.is_tran
$= S_{.\mathbf{2}\cdot\mathbf{2}}.\mathsf{thd}_3$

▷ Prove tran_via : $\{\forall y : S\}\ \{\forall x, z \mid S\}\ (x = y) \to (y = z) \to x = z$
$= [\lambda y : S]\ [\lambda x, z \mid S]\ S_{.\mathbf{2}\cdot\mathbf{2}}.\mathsf{thd}_3 | x | y | z$

▷ Discharge S

B.2.2 Mappings

B.2.2.1 Unary and binary mappings

▷ Introduce S_1, S_2, S_3

▷ Define is_map $= [\lambda f : S_1 \to S_2]\ \{\forall x_1, x_2 \mid S_1\}\ (x_1 = x_2) \to (f\ x_1) = (f\ x_2)$
: $(S_1 \to S_2) \to$ prop

▷ Define is_map$_2$ $= [\lambda f : S_1 \to S_2 \to S_3]\ \{\forall x_1, x_2 \mid S_1\}\ (x_1 = x_2) \to$
$\{\forall y_1, y_2 \mid S_2\}\ (y_1 = y_2) \to (f\ x_1\ y_1) = (f\ x_2\ y_2)$: $(S_1 \to S_2 \to S_3) \to$ prop

▷ Discharge S_3, S_2, S_1

▷ Introduce S_1, S_2, S_3 : set

▷ Let map_el $= \langle \Sigma f : S_1 \to S_2 \rangle\ f$.is_map : Type_0

▷ Let π-coercion ap $= [\lambda f :$ map_el$]\ f_{.\mathbf{1}}$: map_el $\to S_1 \to S_2$

▷ Let map_eq $= [\lambda f_1, f_2 :$ map_el$]\ \{\forall x : S_1\}\ (f_1\ x) = (f_2\ x)$: rel map_el

▷ Prove subresult map_eq_refl : map_eq.is_refl
$= [\lambda f :$ map_el$]\ [\lambda x : S_1]$ refl $(f\ x)$

▷ Prove subresult map_eq_symm : map_eq.is_symm
= $[\lambda f_1, f_2 \,|\, \mathsf{map\_el}]\ [\lambda Q : \mathsf{map\_eq}\ f_1\ f_2]\ [\lambda x : S_1]\ \mathsf{symm}\ (Q\ x)$

▷ Prove subresult map_eq_tran : map_eq.is_tran
= $[\lambda f_1, f_2, f_3 \,|\, \mathsf{map\_el}]\ [\lambda Q_1 : \mathsf{map\_eq}\ f_1\ f_2]\ [\lambda Q_2 : \mathsf{map\_eq}\ f_2\ f_3]\ [\lambda x : S_1]$
$\mathsf{tran}\ (Q_1\ x)\ (Q_2\ x)$

▷ Prove map_forms_set : set_axioms map_eq
= pair_3 map_eq_refl map_eq_symm map_eq_tran

▷ Define map = $(\langle \Sigma f : S_1 \to S_2 \rangle\ f.\mathsf{is\_map},$
$[\lambda f_1, f_2 : \langle \Sigma f : S_1 \to S_2 \rangle\ f.\mathsf{is\_map}]\ \{\forall x : S_1\}\ (f_1.\mathsf{ap}\ x) = (f_2.\mathsf{ap}\ x),$
map_forms_set : set) : set

▷ Let $\mathsf{map}_2\mathsf{\_el} = \langle \Sigma f : S_1 \to S_2 \to S_3 \rangle\ f.\mathsf{is\_map}_2$: Type_0

▷ Let π-coercion $\mathsf{ap}_2 = [\lambda f : \mathsf{map}_2\mathsf{\_el}]\ f_{.\mathbf{1}}$: $\mathsf{map}_2\mathsf{\_el} \to S_1 \to S_2 \to S_3$

▷ Let $\mathsf{map}_2\mathsf{\_eq} = [\lambda f_1, f_2 : \mathsf{map}_2\mathsf{\_el}]\ \{\forall x : S_1\}\ \{\forall y : S_2\}\ (f_1\ x\ y) = (f_2\ x\ y)$
: rel map_2_el

▷ Prove subresult map_2_eq_refl : map_2_eq.is_refl
= $[\lambda f : \mathsf{map}_2\mathsf{\_el}]\ [\lambda x : S_1]\ [\lambda y : S_2]\ \mathsf{refl}\ (f\ x\ y)$

▷ Prove subresult map_2_eq_symm : map_2_eq.is_symm
= $[\lambda f_1, f_2 \,|\, \mathsf{map}_2\mathsf{\_el}]\ [\lambda Q : \mathsf{map}_2\mathsf{\_eq}\ f_1\ f_2]\ [\lambda x : S_1]\ [\lambda y : S_2]\ \mathsf{symm}\ (Q\ x\ y)$

▷ Prove subresult map_2_eq_tran : map_2_eq.is_tran
= $[\lambda f_1, f_2, f_3 \,|\, \mathsf{map}_2\mathsf{\_el}]\ [\lambda Q_1 : \mathsf{map}_2\mathsf{\_eq}\ f_1\ f_2]\ [\lambda Q_2 : \mathsf{map}_2\mathsf{\_eq}\ f_2\ f_3]\ [\lambda x : S_1]$
$[\lambda y : S_2]\ \mathsf{tran}\ (Q_1\ x\ y)\ (Q_2\ x\ y)$

▷ Prove map_2_forms_set : set_axioms map_2_eq
= pair_3 map_2_eq_refl map_2_eq_symm map_2_eq_tran

▷ Define map_2
= $(\langle \Sigma f : S_1 \to S_2 \to S_3 \rangle\ f.\mathsf{is\_map}_2, [\lambda f_1, f_2 : \langle \Sigma f : S_1 \to S_2 \to S_3 \rangle\ f.\mathsf{is\_map}_2]$
$\{\forall x : S_1\}\ \{\forall y : S_2\}\ (f_1.\mathsf{ap}_2\ x\ y) = (f_2.\mathsf{ap}_2\ x\ y), \mathsf{map}_2\mathsf{\_forms\_set} : \mathsf{set})$: set

▷ Discharge map_2_eq_tran, map_2_eq_symm, map_2_eq_refl, map_2_eq, ap_2, map_2_el,
map_eq_tran, map_eq_symm, map_eq_refl, map_eq, ap, map_el, S_3, S_2, S_1

▷ Introduce S_1, S_2, S_3

▷ Define π-coercion ap = $[\lambda f : \mathsf{map}\ S_1\ S_2]\ f_{.\mathbf{1}}$: $(\mathsf{map}\ S_1\ S_2) \to S_1 \to S_2$

▷ Define π-coercion $\mathsf{ap}_2 = [\lambda f : \mathsf{map}_2\ S_1\ S_2\ S_3]\ f_{.\mathbf{1}}$
: $(\mathsf{map}_2\ S_1\ S_2\ S_3) \to S_1 \to S_2 \to S_3$

▷ Introduce g : $\mathsf{map}_2\ S_1\ S_2\ S_3$ and f : $\mathsf{map}\ S_1\ S_2$

▷ Introduce $x_1, x_2 \mid S_1$; suppose Qx : $x_1 = x_2$; introduce x : S_1; y : S_2; $y_1, y_2 \mid S_2$ and suppose Qy : $y_1 = y_2$

▷ Prove resp : $(f\ x_1) = (f\ x_2)$
$= f.\mathbf{2}\ Qx$

▷ Prove resps : $(f\ x_1) = (f\ x_2)$
$= f.\mathbf{2}\ Qx$

▷ Prove resp_2 : $(g\ x_1\ y_1) = (g\ x_2\ y_2)$
$= g.\mathbf{2}\ Qx\ Qy$

▷ Prove resps_1 : $(g\ x_1\ y) = (g\ x_2\ y)$
$= g.\mathbf{2}\ Qx\ (\mathsf{refl}\ y)$

▷ Prove resps_2 : $(g\ x\ y_1) = (g\ x\ y_2)$
$= g.\mathbf{2}\ (\mathsf{refl}\ x)\ Qy$

▷ Discharge $Qy, y_2, y_1, y, x, Qx, x_2, x_1, f, g, S_3, S_2, S_1$

B.2.2.2 Composition and identity for mappings

▷ Introduce S, S_1, S_2, S_3

▷ Prove $\mathsf{identity\_forms\_map}$: $([\lambda x : S]\ x).\mathsf{is\_map}$
$= [\lambda x_1, x_2 \mid S]\ [\lambda H : x_1 = x_2]\ H$

▷ Define $\mathsf{identity} = ([\lambda x : S]\ x, \mathsf{identity\_forms\_map} : \mathsf{map}\ S\ S)$: $\mathsf{map}\ S\ S$

▷ Prove $\mathsf{compose\_makes\_map}$
: $\{\forall f : \mathsf{map}\ S_2\ S_3\}\ \{\forall g : \mathsf{map}\ S_1\ S_2\}\ ([\lambda x : S_1]\ f\ (g\ x)).\mathsf{is\_map}$
$= [\lambda f : \mathsf{map}\ S_2\ S_3]\ [\lambda g : \mathsf{map}\ S_1\ S_2]\ [\lambda x_1, x_2 \mid S_1]\ [\lambda Qx : x_1 = x_2]$
$f.\mathsf{resp}\ (g.\mathsf{resp}\ Qx)$

▷ Prove $\mathsf{compose\_forms\_map}_2$: $\{\forall f_1, f_2 \mid \mathsf{map}\ S_2\ S_3\}\ (f_1 = f_2) \to \{\forall g_1, g_2 \mid \mathsf{map}\ S_1\ S_2\}\ (g_1 = g_2) \to \{\forall x : S_1\}\ (f_1\ (g_1\ x)) = (f_2\ (g_2\ x))$
$= [\lambda f_1, f_2 \mid \mathsf{map}\ S_2\ S_3]\ [\lambda Qf : f_1 = f_2]\ [\lambda g_1, g_2 \mid \mathsf{map}\ S_1\ S_2]\ [\lambda Qg : g_1 = g_2]$
$[\lambda x : S_1]\ (Qf\ (g_1\ x)).\mathsf{tran}\ (f_2.\mathsf{resp}\ (Qg\ x))$

▷ Allow $\circ$ to be written infix

▷ Define $\circ = ([\lambda f : \mathsf{map}\ S_2\ S_3]\ [\lambda g : \mathsf{map}\ S_1\ S_2]$
$([\lambda x : S_1]\ f\ (g\ x), \mathsf{compose\_makes\_map}\ f\ g : \mathsf{map}\ S_1\ S_3)$,
$\mathsf{compose\_forms\_map}_2 : \mathsf{map}_2\ (\mathsf{map}\ S_2\ S_3)\ (\mathsf{map}\ S_1\ S_2)\ (\mathsf{map}\ S_1\ S_3))$
: $\mathsf{map}_2\ (\mathsf{map}\ S_2\ S_3)\ (\mathsf{map}\ S_1\ S_2)\ (\mathsf{map}\ S_1\ S_3)$

▷ Discharge S_3, S_2, S_1, S

B.2.2.3 Results concerning composition and identity for mappings

▷ Introduce S_0, S_1, S_2, S_3

▷ Introduce $f_1, f_2 \mid \mathsf{map}\ S_2\ S_3$ and suppose $Qf : f_1 = f_2$

▷ Introduce $f : \mathsf{map}\ S_2\ S_3$; $g : \mathsf{map}\ S_1\ S_2$ and $h : \mathsf{map}\ S_0\ S_1$

▷ Introduce $g_1, g_2 \mid \mathsf{map}\ S_1\ S_2$ and suppose $Qg : g_1 = g_2$

▷ Prove $\mathsf{compose\_resp} : (f_1 \circ g_1) = (f_2 \circ g_2)$
$= \circ.\mathsf{resp}_2\ Qf\ Qg$

▷ Prove $\mathsf{compose}_1 : (f_1 \circ g) = (f_2 \circ g)$
$= \circ.\mathsf{resps}_1\ Qf\ g$

▷ Prove $\mathsf{compose}_2 : (f \circ g_1) = (f \circ g_2)$
$= \circ.\mathsf{resps}_2\ f\ Qg$

▷ Prove $\mathsf{rewrite\_compose} : \{\forall x : S_1\}\ (f \circ g\ x) = (f\ (g\ x))$
$= [\lambda x : S_1]\ \mathsf{refl}\ (f\ (g\ x))$

▷ Prove $\mathsf{compose\_assoc} : ((f \circ g) \circ h) = (f \circ (g \circ h))$
$= [\lambda x : S_0]\ \mathsf{refl}\ (f\ (g\ (h\ x)))$

▷ Prove $\mathsf{identity\_compose} : (\mathsf{identity} \circ g) = g$
$= [\lambda x : S_1]\ \mathsf{refl}\ (g\ x)$

▷ Prove $\mathsf{compose\_identity} : (g \circ \mathsf{identity}) = g$
$= [\lambda x : S_1]\ \mathsf{refl}\ (g\ x)$

▷ Discharge $Qg, g_2, g_1, h, g, f, Qf, f_2, f_1, S_3, S_2, S_1, S_0$

B.2.3 Subsets

B.2.3.1 Subset definitions

▷ Define $\mathsf{subset\_axioms} = [\lambda S]\ [\lambda A : S \to \mathsf{prop}]\ \{\forall x_1, x_2 \mid S\}\ (x_1 = x_2) \to (A\ x_1) \to A\ x_2 : \{\Pi S\}\ (S \to \mathsf{prop}) \to \mathsf{prop}$

▷ Define $\mathsf{subset} = [\lambda S : \mathsf{set}]\ \langle \Sigma A : S \to \mathsf{prop} \rangle\ \mathsf{subset\_axioms}\ A : \mathsf{set} \to \mathsf{Type}_1$

▷ Introduce S

▷ Unless otherwise specified, by default $A, B, A_1, A_2, A_3 \mid \mathsf{subset}\ S$

▷ Introduce x **:** S and A **:** subset S

▷ Define pred $= A_{.\mathbf{1}}$ **:** $S \to$ prop

▷ Allow $\in$ to be written infix

▷ Define $\in$ $=$ pred x **:** prop

▷ Discharge A, x

▷ Introduce A **:** subset S

▷ Prove subresult subset_eq_refl **:** $([\lambda x, y : \langle \Sigma x' : S \rangle\ x' \in A]\ x_{.\mathbf{1}} = y_{.\mathbf{1}})$.is_refl
$= [\lambda x : \langle \Sigma x : S \rangle\ x \in A]$ refl $x_{.\mathbf{1}}$

▷ Prove subresult subset_eq_symm **:** $([\lambda x, y : \langle \Sigma x' : S \rangle\ x' \in A]\ x_{.\mathbf{1}} = y_{.\mathbf{1}})$.is_symm
$= [\lambda x, y \mid \langle \Sigma x' : S \rangle\ x' \in A]$ symm$|S|x_{.\mathbf{1}}|y_{.\mathbf{1}}$

▷ Prove subresult subset_eq_tran **:** $([\lambda x, y : \langle \Sigma x' : S \rangle\ x' \in A]\ x_{.\mathbf{1}} = y_{.\mathbf{1}})$.is_tran
$= [\lambda x, y, z \mid \langle \Sigma x' : S \rangle\ x' \in A]$ tran$|S|x_{.\mathbf{1}}|y_{.\mathbf{1}}|z_{.\mathbf{1}}$

▷ Prove subset_forms_set **:** set_axioms $([\lambda x, y : \langle \Sigma x' : S \rangle\ x' \in A]\ x_{.\mathbf{1}} = y_{.\mathbf{1}})$
$=$ pair$_3$ subset_eq_refl subset_eq_symm subset_eq_tran

▷ Discharge subset_eq_tran, subset_eq_symm, subset_eq_refl, A

▷ Define coercion as_a_set $= [\lambda A :$ subset $S]$

$(\langle \Sigma x : S \rangle\ x \in A, [\lambda x, y : \langle \Sigma x' : S \rangle\ x' \in A]\ x_{.\mathbf{1}} = y_{.\mathbf{1}}$, subset_forms_set A : set)

: (subset S) $\to$ set

▷ Define as_el_of $= [\lambda x : S]\ [\lambda A :$ subset $S]\ [\lambda xELA : x \in A]\ (x, xELA : A)$

: $\{\Pi x : S\}\ \{\Pi A :$ subset $S\}\ (x \in A) \to A$

▷ Introduce A **:** subset S

▷ Prove rep_forms_map **:** $\{\forall x_1, x_2 \mid A\}\ (x_1.$(equal_in A) $x_2) \to x_{1\cdot\mathbf{1}} = x_{2\cdot\mathbf{1}}$
$= [\lambda x_1, x_2 \mid A]\ [\lambda Q : x_1.$(equal_in A) $x_2]\ Q$

▷ Define rep_map $= ([\lambda x : A]\ x_{.\mathbf{1}}$, rep_forms_map : map $A\ S)$ **:** map $A\ S$

▷ Define coercion rep $=$ rep_map.ap **:** $A \to S$

▷ Discharge A

▷ Introduce A

▷ Prove eq_closed **:** $\{\forall x_1, x_2 \mid S\}\ (x_1 = x_2) \to (x_1 \in A) \to x_2 \in A$
$= [\lambda x_1, x_2 \mid S]\ [\lambda Q : x_1 = x_2]\ [\lambda X_1 : x_1 \in A]\ A_{.\mathbf{2}}\ Q\ X_1$

▷ Define ev $= [\lambda x : A]\ x_{.\mathbf{2}}$ **:** $\{\Pi x : A\}\ x \in A$

▷ Define coercion $\mathsf{make} = [\lambda x \,|\, S]\ x.\mathsf{as\_el\_of}\ A : \{\Pi x \,|\, S\}\ (x \in A) \to A$

▷ Discharge A

▷ Allow $\subseteq$ to be written infix

▷ Define $\subseteq = [\lambda A, B : \mathsf{subset}\ S]\ \{\forall x : A\}\ x \in B$

$: (\mathsf{subset}\ S) \to (\mathsf{subset}\ S) \to \mathsf{prop}$

▷ Introduce A, B and suppose $H : A \subseteq B$

▷ Prove $\mathsf{inclusion\_forms\_map}$
$: \{\forall x, y \,|\, A\}\ (x = y) \to (\mathsf{make}\ (H\ x.\mathsf{ev})) = (\mathsf{make}\ (H\ y.\mathsf{ev}))$
$= [\lambda x, y \,|\, A]\ [\lambda Q : x = y]\ Q$

▷ Define $\mathsf{inclusion} = ([\lambda x : A]\ H\ x.\mathsf{ev}, \mathsf{inclusion\_forms\_map} : \mathsf{map}\ A\ B)$

$: \mathsf{map}\ A\ B$

▷ Discharge H, B, A

▷ Introduce $A, B : \mathsf{subset}\ S$

▷ Explicitly overload the identifier $=$

▷ Define $= = (A \subseteq B) \wedge (B \subseteq A) : \mathsf{prop}$

▷ Prove $\mathsf{subs\_refl} : A \subseteq A$
$= [\lambda x : A]\ x.\mathsf{ev}$

▷ Discharge B, A

▷ Prove $\mathsf{subs\_tran} : \{\forall A_1, A_2, A_3\}\ (A_1 \subseteq A_2) \to (A_2 \subseteq A_3) \to A_1 \subseteq A_3$
$= [\lambda A_1, A_2, A_3]\ [\lambda H_1 : A_1 \subseteq A_2]\ [\lambda H_2 : A_2 \subseteq A_3]\ [\lambda x : A_1]\ H_2\ (H_1\ x)$

▷ Prove $\mathsf{equal\_subs\_refl} : \{\forall A : \mathsf{subset}\ S\}\ A = A$
$= [\lambda A : \mathsf{subset}\ S]\ \mathsf{pair}\ (\mathsf{subs\_refl}\ A)\ (\mathsf{subs\_refl}\ A)$

▷ Prove $\mathsf{equal\_subs\_symm} : \{\forall A_1, A_2\}\ (A_1 = A_2) \to A_2 = A_1$
$= [\lambda A_1, A_2]\ [\lambda H : A_1 = A_2]\ \mathsf{pair}\ H.\mathsf{snd}\ H.\mathsf{fst}$

▷ Prove $\mathsf{equal\_subs\_tran} : \{\forall A_1, A_2, A_3\}\ (A_1 = A_2) \to (A_2 = A_3) \to A_1 = A_3$
$= [\lambda A_1, A_2, A_3]\ [\lambda H_1 : A_1 = A_2]\ [\lambda H_2 : A_2 = A_3]$
$\mathsf{pair}\ (\mathsf{subs\_tran}\ H_1.\mathsf{fst}\ H_2.\mathsf{fst})\ (\mathsf{subs\_tran}\ H_2.\mathsf{snd}\ H_1.\mathsf{snd})$

▷ Define $\mathsf{subset\_of} = [\lambda A : \mathsf{subset}\ S]\ \langle \Sigma B : \mathsf{subset}\ S \rangle\ B \subseteq A$

$: (\mathsf{subset}\ S) \to \mathsf{Type}_1$

▷ Introduce $A, B : \mathsf{subset}\ S$

▷ Define coercion $\mathsf{unsubs} = [\lambda B : \mathsf{subset\_of}\ A]\ B.\mathbf{1} : (\mathsf{subset\_of}\ A) \to \mathsf{subset}\ S$

▷ Define coercion $\mathsf{make\_subs} = [\lambda H : B \subseteq A]\ (B, H : \mathsf{subset\_of}\ A)$

$: (B \subseteq A) \to \mathsf{subset\_of}\ A$

▷ Prove $\mathsf{intersect\_forms\_subset} : \mathsf{subset\_axioms}\ ([\lambda x : S]\ (x \in A) \wedge (x \in B))$

$= [\lambda x_1, x_2 \mid S]\ [\lambda Q x : x_1 = x_2]\ [\lambda H : (x_1 \in A) \wedge (x_1 \in B)]$

$\mathsf{pair}\ (\mathsf{eq\_closed}\ Qx\ H.\mathsf{fst})\ (\mathsf{eq\_closed}\ Qx\ H.\mathsf{snd})$

▷ Allow $\cap$ to be written infix

▷ Define $\cap = ([\lambda x : S]\ (x \in A) \wedge (x \in B), \mathsf{intersect\_forms\_subset} : \mathsf{subset}\ S)$

$: \mathsf{subset}\ S$

▷ Discharge B, A, S

B.2.3.2 Placing one subset inside another

▷ Introduce S and $B, A : \mathsf{subset}\ S$

▷ Prove $\mathsf{aaso\_forms\_subset} : \mathsf{subset\_axioms}|A\ ([\lambda x : A]\ x.\mathsf{rep} \in B)$

$= [\lambda x_1, x_2 \mid A]\ [\lambda Q : x_1 = x_2]\ [\lambda H : x_1.\mathsf{rep} \in B]\ \mathsf{eq\_closed}|S|B\ Q\ H$

▷ Define $\mathsf{as\_a\_subset\_of} = ([\lambda x : A]\ x.\mathsf{rep} \in B, \mathsf{aaso\_forms\_subset} : \mathsf{subset}\ A)$

$: \mathsf{subset}\ A$

▷ Discharge A, B

▷ Introduce $A : \mathsf{subset}\ S$; $B_1, B_2 \mid \mathsf{subset}\ S$; suppose $H : B_1 \subseteq B_2$ and Q

$: B_1 = B_2$

▷ Prove $\mathsf{AASO_1s} : (B_1.\mathsf{as\_a\_subset\_of}\ A) \subseteq (B_2.\mathsf{as\_a\_subset\_of}\ A)$

$= [\lambda x : B_1.\mathsf{as\_a\_subset\_of}\ A]\ H\ (x.\mathsf{rep.rep.as\_el\_of}\ B_1\ x.\mathsf{ev})$

▷ Discharge but keep Q, H, B_2, B_1

▷ Prove $\mathsf{AASO_1} : (B_1.\mathsf{as\_a\_subset\_of}\ A) = (B_2.\mathsf{as\_a\_subset\_of}\ A)$

$= \mathsf{pair}\ (\mathsf{AASO_1s}\ Q.\mathsf{fst})\ (\mathsf{AASO_1s}\ Q.\mathsf{snd})$

▷ Discharge Q, H, B_2, B_1, A, S

B.2.3.3 Singleton subsets

▷ Introduce S and $e : S$

▷ Construct by refinement $\mathsf{singleton\_forms\_subset} : \mathsf{subset\_axioms}\ e =$

$(\mathsf{tran}, =)$

▷ Define $\mathsf{singleton} = (e =, \mathsf{singleton\_forms\_subset} : \mathsf{subset}\ S) : \mathsf{subset}\ S$

▷ Discharge e

▷ Construct by refinement $\mathsf{singleton_1s}$
$: \{\Pi e_1, e_2 \mid S\}\ (e_1 = e_2) \to (\mathsf{singleton}\ e_1) \subseteq (\mathsf{singleton}\ e_2)$
(singleton, ev, symm, tran, =)

▷ Construct by refinement $\mathsf{singleton_1}$
$: \{\Pi e_1, e_2 \mid S\}\ (e_1 = e_2) \to (\mathsf{singleton}\ e_1) = (\mathsf{singleton}\ e_2)$
(symm, $\mathsf{singleton_1s}$, singleton, $\subseteq$, pair, =)

▷ Discharge S

B.2.4 Quotients

B.2.4.1 Equivalence relations

▷ Define $\mathsf{is\_equiv\_rel} = [\lambda S]\ [\lambda\sim : \mathsf{rel}\ S]\ \bigwedge_3\ (\{\Pi x, y \mid S\}\ (x = y) \to x \sim y)$
$\sim.\mathsf{is\_symm}\ \sim.\mathsf{is\_tran} : \{\Pi S\}\ (\mathsf{rel}\ S) \to \mathsf{prop}$

▷ Define $\mathsf{equiv\_rel} = [\lambda S : \mathsf{set}]\ \langle \Sigma\sim : \mathsf{rel}\ S\rangle\ \sim.\mathsf{is\_equiv\_rel} : \mathsf{set} \to \mathsf{Type}_1$

▷ Define π-coercion $\mathsf{ap\_equiv\_rel} = [\lambda S]\ [\lambda\sim : \mathsf{equiv\_rel}\ S]\ \sim._{\mathbf{1}}$
$: \{\Pi S\}\ (\mathsf{equiv\_rel}\ S) \to \mathsf{rel}\ S$

B.2.4.2 Equality as an equivalence relation

▷ Construct by refinement $\mathsf{eq\_is\_eqrel} : \{\Pi S : \mathsf{set}\}\ (\mathsf{equal\_in}\ S).\mathsf{is\_equiv\_rel}$
(tran, symm, =, equal_in, is_tran, is_symm, $\mathsf{pair_3}$, set)

▷ Explicitly overload the identifier =

▷ Define $= = [\lambda S]\ (\mathsf{equal\_in}\ S, \mathsf{eq\_is\_eqrel}\ S : \mathsf{equiv\_rel}\ S) : \{\Pi S\}\ \mathsf{equiv\_rel}\ S$

B.2.4.3 The quotient of a set

▷ Introduce $S : \mathsf{set}$ and $\sim : \mathsf{equiv\_rel}\ S$

▷ Prove $\mathsf{quotient\_forms\_set} : \mathsf{set\_axioms}\ \sim$
$= \mathsf{pair_3}\ ([\lambda x : S]\ \sim._{\mathbf{2}}.\mathsf{fst_3}\ (\mathsf{refl}\ x))\ \sim._{\mathbf{2}}.\mathsf{snd_3}\ \sim._{\mathbf{2}}.\mathsf{thd_3}$

▷ Allow / to be written midfix

▷ Define $/ = (S, \sim, \mathsf{quotient\_forms\_set} : \mathsf{set}) : \mathsf{set}$

▷ Discharge $\sim, S$

B.2.4.4 The quotient of a subset

▷ Introduce S; A : subset S and $\sim$: equiv_rel S

▷ Define $\mathsf{is\_quotsubs} = [\lambda x : S/\sim]\ \langle \Sigma y : A \rangle\ y \sim x : (S/\sim) \to \mathsf{prop}$

▷ Prove $\mathsf{quotsubs\_forms\_subset} : \mathsf{subset\_axioms}\ \mathsf{is\_quotsubs}$
$= [\lambda x_1, x_2 \,|\, S/\sim]\ [\lambda Q : x_1 = x_2]\ [\lambda H : x_1.\mathsf{is\_quotsubs}]$
$(H.\mathbf{1}, \mathsf{tran}|(S/\sim)\ H.\mathbf{2}\ Q : x_2.\mathsf{is\_quotsubs})$

▷ Explicitly overload the identifier $/$

▷ Define $/ = (\mathsf{is\_quotsubs}, \mathsf{quotsubs\_forms\_subset} : \mathsf{subset}\ (S/\sim))$
$: \mathsf{subset}\ (S/\sim)$

▷ Discharge $\sim, A, S$

B.2.5 Set morphisms

B.2.5.1 Split set monomorphisms (left-invertible mappings)

▷ Introduce S, T

▷ Define $\mathsf{is\_split\_mono} = [\lambda f : \mathsf{map}\ S\ T]\ \langle \exists f' : \mathsf{map}\ T\ S \rangle\ (f' \circ f) = \mathsf{identity}$
$: (\mathsf{map}\ S\ T) \to \mathsf{prop}$

▷ Prove $\mathsf{split\_mono\_forms\_subset} : \mathsf{subset\_axioms}\ \mathsf{is\_split\_mono}$
$= [\lambda f_1, f_2 \,|\, \mathsf{map}\ S\ T]\ [\lambda Q f : f_1 = f_2]\ [\lambda H : f_1.\mathsf{is\_split\_mono}]$
$(H.\mathbf{1}, (\mathsf{compose}_2\ H.\mathbf{1}\ Qf.\mathsf{symm}).\mathsf{tran}\ H.\mathbf{2} : f_2.\mathsf{is\_split\_mono})$

▷ Discharge T, S

▷ Define $\mathsf{split\_mono} = [\lambda S, T : \mathsf{set}]$
$(\mathsf{is\_split\_mono}|S|T, \mathsf{split\_mono\_forms\_subset}|S|T : \mathsf{subset}\ (\mathsf{map}\ S\ T))$
$: \{\Pi S, T : \mathsf{set}\}\ \mathsf{subset}\ (\mathsf{map}\ S\ T)$

B.2.5.2 Split set epimorphisms (right-invertible mappings)

▷ Introduce S, T

▷ Define is_split_epi $= [\lambda f : \mathsf{map}\ S\ T]\ \langle \exists f' : \mathsf{map}\ T\ S \rangle\ (f \circ f') = \mathsf{identity}$

$:\ (\mathsf{map}\ S\ T) \to \mathsf{prop}$

▷ Prove split_epi_forms_subset : subset_axioms is_split_epi

$= [\lambda f_1, f_2 \mid \mathsf{map}\ S\ T]\ [\lambda Q f : f_1 = f_2]\ [\lambda H : f_1.\mathsf{is\_split\_epi}]$
$(H.\mathbf{1}, (\mathsf{compose}_1\ Qf.\mathsf{symm}\ H.\mathbf{1}).\mathsf{tran}\ H.\mathbf{2} : f_2.\mathsf{is\_split\_epi})$

▷ Discharge T, S

▷ Define split_epi $= [\lambda S, T : \mathsf{set}]$

$(\mathsf{is\_split\_epi}|S|T, \mathsf{split\_epi\_forms\_subset}|S|T : \mathsf{subset}\ (\mathsf{map}\ S\ T))$

$:\ \{\Pi S, T : \mathsf{set}\}\ \mathsf{subset}\ (\mathsf{map}\ S\ T)$

B.2.5.3 Set isomorphisms

▷ Introduce S, T

▷ Define is_iso $= [\lambda f : \mathsf{map}\ S\ T]\ \langle \exists f' : \mathsf{map}\ T\ S \rangle$

$((f' \circ f) = \mathsf{identity}) \wedge ((f \circ f') = \mathsf{identity})\ :\ (\mathsf{map}\ S\ T) \to \mathsf{prop}$

▷ Prove iso_forms_subset : subset_axioms is_iso

$= [\lambda f_1, f_2 \mid \mathsf{map}\ S\ T]\ [\lambda Q f : f_1 = f_2]\ [\lambda H : f_1.\mathsf{is\_iso}]$
$(H.\mathbf{1}, \mathsf{pair}\ ((H.\mathbf{1}.\mathsf{compose}_2\ Qf).\mathsf{symm.tran}\ H.\mathbf{2}.\mathsf{fst})$
$((Qf.\mathsf{compose}_1\ H.\mathbf{1}).\mathsf{symm.tran}\ H.\mathbf{2}.\mathsf{snd}) : f_2.\mathsf{is\_iso})$

▷ Discharge T, S

▷ Define iso

$= [\lambda S, T : \mathsf{set}]\ (\mathsf{is\_iso}|S|T, \mathsf{iso\_forms\_subset}|S|T : \mathsf{subset}\ (\mathsf{map}\ S\ T))$

$:\ \{\Pi S, T : \mathsf{set}\}\ \mathsf{subset}\ (\mathsf{map}\ S\ T)$

▷ Introduce S, T

▷ Prove inverse_makes_iso : $\{\Pi f : \mathsf{iso}\ S\ T\}\ (f.\mathsf{ev}).\mathbf{1} \in (\mathsf{iso}\ T\ S)$

$= [\lambda f : \mathsf{iso}\ S\ T]\ (f.\mathsf{rep}, \mathsf{pair}\ (f.\mathsf{ev}).\mathbf{2}.\mathsf{snd}\ (f.\mathsf{ev}).\mathbf{2}.\mathsf{fst} : (f.\mathsf{ev}).\mathbf{1} \in (\mathsf{iso}\ T\ S))$

▷ Introduce $f \mid \mathsf{map}\ S\ T$ and $f_1, f_2 \mid \mathsf{map}\ T\ S$

▷ Suppose $Qf_1 : (f_1 \circ f) = \mathsf{identity}$ and $Qf_2 : (f \circ f_2) = \mathsf{identity}$

▷ Prove subresult $\mathsf{Q}_1 : f_1 = (f_1 \circ \mathsf{identity})$

$= f_1.\mathsf{compose\_identity.symm}$

▷ Prove subresult $\mathsf{Q}_2 : (f_1 \circ \mathsf{identity}) = (f_1 \circ (f \circ f_2))$

$= f_1.\mathsf{compose}_2\ Qf_2.\mathsf{symm}$

▷ Prove subresult Q_3 : $(f_1 \circ (f \circ f_2)) = ((f_1 \circ f) \circ f_2)$
= (compose_assoc f_1 f f_2).symm

▷ Prove subresult Q_4 : $((f_1 \circ f) \circ f_2) = (\mathsf{identity} \circ f_2)$
= Qf_1.compose$_1$ f_2

▷ Prove subresult Q_5 : $(\mathsf{identity} \circ f_2) = f_2$
= identity_compose f_2

▷ Prove inverse_lemma : $f_1 = f_2$
= (Q_1.tran Q_2).tran (Q_3.tran (Q_4.tran Q_5))

▷ Discharge $\mathsf{Q}_5, \mathsf{Q}_4, \mathsf{Q}_3, \mathsf{Q}_2, \mathsf{Q}_1, Qf_2, Qf_1, f_2, f_1, f$

▷ Prove inverse_forms_map
: $([\lambda f : \mathsf{iso}\ S\ T]$ make f.inverse_makes_iso).(is_map|(iso S T)|(iso T S))
= $[\lambda f_1, f_2 \mid \mathsf{iso}\ S\ T]\ [\lambda Qf : f_1 = f_2]$
inverse_lemma (f_1.ev).$_{\mathbf{2}}$.fst ((Qf.compose$_1$ (f_2.ev).$_{\mathbf{1}}$).tran (f_2.ev).$_{\mathbf{2}}$.snd)

▷ Allow $^{-1}$ to be written postfix

▷ Define $^{-1} = ([\lambda f : \mathsf{iso}\ S\ T]$ make f.inverse_makes_iso, inverse_forms_map :
map (iso S T) (iso T S)) : map (iso S T) (iso T S)

▷ Introduce f : iso S T

▷ Prove inverse_compose_iso : $(f^{-1} \circ f) = \mathsf{identity}$
= (f.ev).$_{\mathbf{2}}$.fst

▷ Prove iso_compose_inverse : $(f \circ f^{-1}) = \mathsf{identity}$
= (f.ev).$_{\mathbf{2}}$.snd

▷ Discharge f, T, S

B.2.5.4 Results concerning set morphisms

▷ Introduce S, T

▷ Prove subresult iso_subs_mono : (iso S T) $\subseteq$ (split_mono S T)
= $[\lambda f : \mathsf{iso}\ S\ T]$ (f^{-1}, inverse_compose_iso f : $f \in$ (split_mono S T))

▷ Prove subresult iso_subs_epi : (iso S T) $\subseteq$ (split_epi S T)
= $[\lambda f : \mathsf{iso}\ S\ T]$ (f^{-1}, iso_compose_inverse f : $f \in$ (split_epi S T))

▷ Prove subresult mono_epi_subs_iso
: ((split_mono S T) $\cap$ (split_epi S T)) $\subseteq$ (iso S T)
= $[\lambda f : (\mathsf{split\_mono}\ S\ T) \cap (\mathsf{split\_epi}\ S\ T)]$ ((f.ev.fst).$_{\mathbf{1}}$, pair (f.ev.fst).$_{\mathbf{2}}$
((f.compose$_2$ (inverse_lemma (f.ev.fst).$_{\mathbf{2}}$ (f.ev.snd).$_{\mathbf{2}}$)).tran (f.ev.snd).$_{\mathbf{2}}$) :
$f \in$ (iso S T))

▷ Prove iso_eq_mono_epi : (iso S T) = ((split_mono S T) ∩ (split_epi S T))
= pair ([λf : iso S T] pair (iso_subs_mono f) (iso_subs_epi f)) mono_epi_subs_iso

▷ Prove subresult P : {∀f | map S T} {ΠH_1 : f ∈ (split_mono S T)}
{ΠH_2 : f ∈ (split_epi S T)} (make (pair H_1 H_2)) ∈ (iso S T)
= [λf | map S T] [λH_1 : f ∈ (split_mono S T)] [λH_2 : f ∈ (split_epi S T)]
iso_eq_mono_epi.snd (make (pair H_1 H_2))

▷ Discharge T, S

▷ Introduce S : set

▷ Prove identity_is_split_mono : identity ∈ (split_mono S S)
= (identity|S, identity.compose_identity : identity ∈ (split_mono S S))

▷ Prove identity_is_split_epi : identity ∈ (split_epi S S)
= (identity|S, identity_compose identity : identity ∈ (split_epi S S))

▷ Prove identity_is_iso : identity ∈ (iso S S)
= P identity_is_split_mono identity_is_split_epi

▷ Discharge S

▷ Introduce S_1, S_2, S_3

▷ Introduce f : split_mono S_2 S_3 and g : split_mono S_1 S_2

▷ Let f' = (f.ev)$_{.\mathbf{1}}$: map S_3 S_2 and g' = (g.ev)$_{.\mathbf{1}}$: map S_2 S_1

▷ Prove subresult P_1 : ((g' ∘ f') ∘ (f ∘ g)) = (g' ∘ (f' ∘ (f ∘ g)))
= compose_assoc g' f' (f ∘ g)

▷ Prove subresult $P_{2\text{-}1}$: (f' ∘ (f ∘ g)) = ((f' ∘ f) ∘ g)
= (compose_assoc f' f g).symm

▷ Prove subresult $P_{2\text{-}2}$: (((f.ev)$_{.\mathbf{1}}$ ∘ f) ∘ g) = (identity ∘ g)
= compose$_1$ (f.ev)$_{.\mathbf{2}}$ g

▷ Prove subresult $P_{2\text{-}3}$: (identity ∘ g) = g
= identity_compose g

▷ Prove subresult P_2 : (g' ∘ (f' ∘ (f ∘ g))) = (g' ∘ g)
= compose$_2$ g' ($P_{2\text{-}1}$.tran ($P_{2\text{-}2}$.tran $P_{2\text{-}3}$))

▷ Prove subresult P_3 : [δf' = (g.ev)$_{.\mathbf{1}}$] (f' ∘ g) = identity
= (g.ev)$_{.\mathbf{2}}$

▷ Prove compose_split_monos_is_split_mono : (f ∘ g) ∈ (split_mono S_1 S_3)
= (g' ∘ f', P_1.tran (P_2.tran P_3) : (f ∘ g) ∈ (split_mono S_1 S_3))

▷ Discharge $P_3, P_2, P_{2\text{-}3}, P_{2\text{-}2}, P_{2\text{-}1}, P_1$, g', f', g, f

▷ Introduce f : split_epi S_2 S_3 and g : split_epi S_1 S_2

▷ Let f' $= (f.\mathsf{ev})_{.\mathbf{1}}$: map S_3 S_2 and g' $= (g.\mathsf{ev})_{.\mathbf{1}}$: map S_2 S_1

▷ Prove subresult P_1 : $(f \circ (g \circ (\mathsf{g'} \circ \mathsf{f'}))) = ((f \circ g) \circ (\mathsf{g'} \circ \mathsf{f'}))$
= (compose_assoc f g (g' ∘ f')).symm

▷ Prove subresult $\mathsf{P}_{2\text{-}1}$: $((g \circ \mathsf{g'}) \circ \mathsf{f'}) = (g \circ (\mathsf{g'} \circ \mathsf{f'}))$
= compose_assoc g g' f'

▷ Prove subresult $\mathsf{P}_{2\text{-}2}$: $((g \circ (g.\mathsf{ev})_{.\mathbf{1}}) \circ \mathsf{f'}) = (\mathsf{identity} \circ \mathsf{f'})$
= $\mathsf{compose}_1$ $(g.\mathsf{ev})_{.\mathbf{2}}$ f'

▷ Prove subresult $\mathsf{P}_{2\text{-}3}$: $(\mathsf{identity} \circ \mathsf{f'}) = \mathsf{f'}$
= identity_compose f'

▷ Prove subresult P_2 : $(f \circ ((g \circ \mathsf{g'}) \circ \mathsf{f'})) = (f \circ \mathsf{f'})$
= $\mathsf{compose}_2$ f ($\mathsf{P}_{2\text{-}1}$.tran ($\mathsf{P}_{2\text{-}2}$.tran $\mathsf{P}_{2\text{-}3}$))

▷ Prove subresult P_3 : $[\delta\mathsf{f'} = (f.\mathsf{ev})_{.\mathbf{1}}]$ $(f \circ \mathsf{f'}) = \mathsf{identity}$
= $(f.\mathsf{ev})_{.\mathbf{2}}$

▷ Prove compose_split_epis_is_split_epi : $(f \circ g) \in (\mathsf{split\_epi}\ S_1\ S_3)$
= (g' ∘ f', P_1.tran (P_2.tran P_3) : $(f \circ g) \in (\mathsf{split\_epi}\ S_1\ S_3)$)

▷ Discharge $\mathsf{P}_3, \mathsf{P}_2, \mathsf{P}_{2\text{-}3}, \mathsf{P}_{2\text{-}2}, \mathsf{P}_{2\text{-}1}, \mathsf{P}_1, \mathsf{g'}, \mathsf{f'}, g, f$

▷ Prove compose_isos_is_iso
: $\{\forall f : \mathsf{iso}\ S_2\ S_3\}\ \{\forall g : \mathsf{iso}\ S_1\ S_2\}\ (f \circ g) \in (\mathsf{iso}\ S_1\ S_3)$
= $[\lambda f : \mathsf{iso}\ S_2\ S_3]\ [\lambda g : \mathsf{iso}\ S_1\ S_2]$
P (compose_split_monos_is_split_mono (iso_subs_mono f) (iso_subs_mono g))
(compose_split_epis_is_split_epi (iso_subs_epi f) (iso_subs_epi g))

▷ Explicitly overload the identifier ∘

▷ Define ∘
$= [\lambda f : \mathsf{iso}\ S_2\ S_3]\ [\lambda g : \mathsf{iso}\ S_1\ S_2]\ (f \circ g, \mathsf{compose\_isos\_is\_iso}\ f\ g : \mathsf{iso}\ S_1\ S_3)$
: $(\mathsf{iso}\ S_2\ S_3) \to (\mathsf{iso}\ S_1\ S_2) \to \mathsf{iso}\ S_1\ S_3$

▷ Prove rewrite_iso_compose
: $\{\forall f : \mathsf{iso}\ S_2\ S_3\}\ \{\forall g : \mathsf{iso}\ S_1\ S_2\}\ \{\forall x : S_1\}\ (f \circ g\ x) = (f\ (g\ x))$
= $[\lambda f : \mathsf{iso}\ S_2\ S_3]\ [\lambda g : \mathsf{iso}\ S_1\ S_2]\ [\lambda x : S_1]$ rewrite_compose $f\ g\ x$

▷ Discharge $S_3, S_2, S_1, \mathsf{P}$, mono_epi_subs_iso, iso_subs_epi, iso_subs_mono

▷ Prove iso_refl : $\{\forall S : \mathsf{set}\}\ \mathsf{iso}\ S\ S$
= $[\lambda S : \mathsf{set}]$ identity_is_iso S

▷ Prove iso_symm : $\{\forall S_1, S_2\}\ (\mathsf{iso}\ S_1\ S_2) \to \mathsf{iso}\ S_2\ S_1$
= $[\lambda S_1, S_2]\ [\lambda H : \mathsf{iso}\ S_1\ S_2]\ H^{-1}$

▷ Prove iso_tran : $\{\forall S_1, S_2, S_3\}$ (iso S_1 S_2) $\rightarrow$ (iso S_2 S_3) $\rightarrow$ iso S_1 S_3
= $[\lambda S_1, S_2, S_3]$ $[\lambda H_1 : \text{iso } S_1\ S_2]$ $[\lambda H_2 : \text{iso } S_2\ S_3]$ compose_isos_is_iso H_2 H_1

▷ Introduce S, T; A_1, A_2; suppose QA : $A_1 = A_2$ and introduce A : subset S

▷ Let iso_to = inclusion QA.fst : map A_1 A_2 and iso_from = inclusion QA.snd : map A_2 A_1

▷ Prove subresult P_1 : $\{\forall x : A_1\}$ (iso_from ∘ iso_to x) = x
= $[\lambda x : A_1]$ refl x

▷ Prove subresult P_2 : $\{\forall y : A_2\}$ (iso_to ∘ iso_from y) = y
= $[\lambda y : A_2]$ refl y

▷ Prove equal_forms_iso : iso_to $\in$ (iso A_1 A_2)
= (iso_from, pair P_1 P_2 : iso_to $\in$ (iso A_1 A_2))

▷ Define equal_iso = (iso_to, equal_forms_iso : iso A_1 A_2) : iso A_1 A_2

▷ Discharge but keep $\mathsf{P}_2, \mathsf{P}_1$, iso_from, iso_to, A, QA, A_2, A_1, T, S

▷ Introduce B_1, B_2 | subset T; suppose QB : $B_1 = B_2$ and introduce B : subset T

▷ Prove ISO_1 : (iso A_1 B) $\rightarrow$ iso A_2 B
= $[\lambda H : \text{iso } A_1\ B]$ (equal_iso QA).iso_symm.iso_tran H

▷ Prove ISO_2 : (iso A B_1) $\rightarrow$ iso A B_2
= $[\lambda H : \text{iso } A\ B_1]$ H.iso_tran (equal_iso QB)

▷ Prove ISOresp : (iso A_1 B_1) $\rightarrow$ iso A_2 B_2
= $[\lambda H : \text{iso } A_1\ B_1]$ (equal_iso QA).iso_symm.iso_tran (H.iso_tran (equal_iso QB))

▷ Discharge $B, QB, B_2, B_1, \mathsf{P}_2, \mathsf{P}_1$, iso_from, iso_to, A, QA, A_2, A_1, T, S

B.3 Decision

B.3.1 Decideability

B.3.1.1 Decideability of a proposition

▷ Define or_not = $[\lambda p]$ $p \vee (\neg p)$: prop $\rightarrow$ prop

B.3.1.2 Alternation

▷ Introduce p_1, p_2 | prop; S and suppose H : $p_1 \vee p_2$

▷ Define if $= [\lambda xt, xf : S]$ case H $([\lambda\_ : p_1]\ xt)$ $([\lambda\_ : p_2]\ xf)$: $S \to S \to S$

▷ Discharge H, S, p_2, p_1

B.3.1.3 Results concerning alternation

▷ Introduce p | prop; S; xt, xf : S and suppose b : p.or_not

▷ Construct by refinement IF_0 : $(\neg p) \to (\text{if } b\ xt\ xf) = xf$

(¬, inr, if, refl, inl, =, ex_falso, or_not, case_elim)

▷ Construct by refinement IF_1 : $p \to (\text{if } b\ xt\ xf) = xt$

(¬, inr, if, =, ex_falso, inl, refl, or_not, case_elim)

▷ Discharge b, xf, xt

▷ Prove IF_2 : $\{\forall xt, xf : S\}\ \{\Pi b_1, b_2 : p.\text{or\_not}\}\ (\text{if } b_1\ xt\ xf) = (\text{if } b_2\ xt\ xf)$

$= [\lambda xt, xf : S]\ [\lambda b_1, b_2 : p.\text{or\_not}]$

case b_1 $([\lambda P : p]$ tran_via xt $(\mathsf{IF}_1\ xt\ xf\ b_1\ P)$ $(\mathsf{IF}_1\ xt\ xf\ b_2\ P)$.symm)

$([\lambda nP : \neg p]$ tran_via xf $(\mathsf{IF}_0\ xt\ xf\ b_1\ nP)$ $(\mathsf{IF}_0\ xt\ xf\ b_2\ nP)$.symm)

▷ Discharge S, p

B.3.1.4 Decideable subsets

▷ Introduce S

▷ Define is_decideable $= [\lambda A : \text{subset } S]\ \{\Pi x : S\}\ (x \in A)$.or_not

: (subset S) $\to$ prop

▷ Define is_decideable_in $= [\lambda A, B : \text{subset } S]\ \{\Pi x : B\}\ (x \in A)$.or_not

: (subset S) $\to$ (subset S) $\to$ prop

▷ Define decideable_subs $= [\lambda A, B : \text{subset } S] \wedge (A.\text{is\_decideable\_in } B)\ (A \subseteq B)$

: (subset S) $\to$ (subset S) $\to$ prop

▷ Discharge S

▷ Define dsubset $= [\lambda S : \text{set}]\ \langle \Sigma A : \text{subset } S \rangle\ A.\text{is\_decideable}$: set $\to$ Type_1

▷ Introduce S

▷ Allow $\notin$ to be written infix

▷ Define $\notin = [\lambda x : S]\ [\lambda A : \mathsf{subset}\ S]\ \neg(x \in A)$ **:** $S \to (\mathsf{subset}\ S) \to \mathsf{prop}$

▷ Define coercion $\mathsf{undecide\_subset} = [\lambda A : \mathsf{dsubset}\ S]\ A.\mathbf{1}$

: $(\mathsf{dsubset}\ S) \to \mathsf{subset}\ S$

▷ Define coercion $\mathsf{make\_dsubset} = [\lambda A]\ [\lambda H : A.\mathsf{is\_decideable}]$

$(A, H : \mathsf{dsubset}\ S)$ **:** $\{\Pi A\}\ A.\mathsf{is\_decideable} \to \mathsf{dsubset}\ S$

▷ Define $\mathsf{decide} = [\lambda A : \mathsf{dsubset}\ S]\ [\lambda x : S]\ A.\mathbf{2}\ x$

: $\{\Pi A : \mathsf{dsubset}\ S\}\ \{\Pi x : S\}\ (x \in A) \vee (x \notin A)$

▷ Allow $\in_?$ to be written infix

▷ Define $\in_? = [\lambda x : S]\ [\lambda A : \mathsf{dsubset}\ S]\ \mathsf{decide}\ A\ x$

: $\{\Pi x : S\}\ \{\Pi A : \mathsf{dsubset}\ S\}\ (x \in A) \vee (x \notin A)$

▷ Discharge S

B.3.1.5 Results concerning decideability

▷ Introduce S, T; C **:** $S \to \mathsf{prop}$ and A **:** $\mathsf{dsubset}\ S$

▷ Suppose H_1 **:** $(\{\forall x : S\}\ (x \in A) \to C\ x) \wedge (\{\forall x : S\}\ (x \notin A) \to C\ x)$

▷ Introduce x **:** S

▷ Prove subresult C_1 **:** $(x \in A) \to C\ x$
= $H_1.\mathsf{fst}\ x$

▷ Prove subresult C_2 **:** $(x \notin A) \to C\ x$
= $H_1.\mathsf{snd}\ x$

▷ Prove subresult P_1 **:** $C\ x$
= $\mathsf{case}\ (x \in_?\ A)\ \mathsf{C}_1\ \mathsf{C}_2$

▷ Discharge x, H_1

▷ Suppose H_2 **:** $\{\forall x : S\}\ C\ x$

▷ Prove subresult $\mathsf{P}_{2\text{-}1}$ **:** $\{\forall x : S\}\ (x \in A) \to C\ x$
= $[\lambda x : S]\ [\lambda \_ : x \in A]\ H_2\ x$

▷ Prove subresult $\mathsf{P}_{2\text{-}2}$ **:** $\{\forall x : S\}\ (x \notin A) \to C\ x$
= $[\lambda x : S]\ [\lambda \_ : x \notin A]\ H_2\ x$

▷ Prove subresult P_2 **:** $(\{\forall x : S\}\ (x \in A) \to C\ x) \wedge (\{\forall x : S\}\ (x \notin A) \to C\ x)$
= $\mathsf{pair}\ \mathsf{P}_{2\text{-}1}\ \mathsf{P}_{2\text{-}2}$

▷ Discharge H_2

▷ Prove DEC_1 : $((\{\Pi x : S\}\ (x \in A) \to C\ x) \wedge (\{\Pi x : S\}\ (x \notin A) \to C\ x)) \leftrightarrow (\{\Pi x : S\}\ C\ x)$
$= \mathsf{pair}\ \mathsf{P}_1\ \mathsf{P}_2$

▷ Discharge $\mathsf{P}_2, \mathsf{P}_{2\_2}, \mathsf{P}_{2\_1}, \mathsf{P}_1, \mathsf{C}_2, \mathsf{C}_1, A, C, T, S$

▷ Construct by refinement DEC_2
: $\{\forall p_1, p_2 \mid \mathsf{prop}\}\ (p_1 \leftrightarrow p_2) \to p_1.\mathsf{or\_not} \to p_2.\mathsf{or\_not}$
(snd, ¬, inr, fst, inl, or_not, case, ↔, prop)

▷ Introduce $p, p_1, p_2 \mid \mathsf{prop}$; H_1 : $p_1.\mathsf{or\_not}$ and H_2 : $p_2.\mathsf{or\_not}$

▷ Construct by refinement DEC_3 : $(p_1 \wedge p_2).\mathsf{or\_not}$
(fst, ∧, ¬, inr, snd, pair, inl, or_not, case)

▷ Construct by refinement DEC_4 : $(p_1 \vee p_2).\mathsf{or\_not}$
(false, case, ∨, ¬, inr, inl, or_not)

▷ Construct by refinement DEC_5 : $(\neg p_1).\mathsf{or\_not}$
(¬, inl, false, inr, or_not, case)

▷ Prove DEC_6 : $p.\mathsf{or\_not} \to (\neg(\neg p)) \to p$
$= [\lambda b : p.\mathsf{or\_not}]\ [\lambda P : \neg(\neg p)]\ \mathsf{case}\ b\ ([\lambda H : p]\ H)\ ([\lambda H : \neg p]\ \mathsf{ex\_falso}\ p\ (P\ H))$

▷ Prove $\mathsf{or\_not\_is\_true}$: $p \to p.\mathsf{or\_not}$
$= \mathsf{inl}|p|(\neg p)$

▷ Prove $\mathsf{or\_not\_is\_false}$: $(\neg p) \to p.\mathsf{or\_not}$
$= \mathsf{inr}|p|(\neg p)$

▷ Discharge H_2, H_1, p_2, p_1, p

B.3.2 Discreteness

B.3.2.1 Discrete sets

▷ Allow $\neq$ to be written infix

▷ Define $\neq = [\lambda S]\ [\lambda x, y : S]\ \neg(x = y)$: $\{\Pi S\}\ \mathsf{rel}\ S$

▷ Define $\mathsf{is\_discrete} = [\lambda S : \mathsf{set}]\ \{\forall x, y : S\}\ (x = y) \vee (x \neq y)$: $\mathsf{set} \to \mathsf{prop}$

B.3.3 Altering sets and subsets

B.3.3.1 Excluding an element from a subset

▷ Introduce S; A : subset S and $a : S$

▷ Prove everything_except_forms_subset : subset_axioms $a \neq$
$= [\lambda x, y \,|\, S]\ [\lambda Q : x = y]\ [\lambda H : a \neq x]\ [\lambda X : a = y]\ H\ (X.\mathsf{tran}\ Q.\mathsf{symm})$

▷ Define everything_except $= (a \neq, \mathsf{everything\_except\_forms\_subset} : \mathsf{subset}\ S)$
: subset S

▷ Discharge a

▷ Introduce $a : A$

▷ Allow $\setminus$ to be written infix

▷ Define $\setminus = A \cap (\mathsf{everything\_except}\ a)$: subset S

▷ Define coercion unexclude $= [\lambda x : \setminus]$ make $x.\mathsf{ev.fst} : \setminus \to A$

▷ Discharge a, A, S

B.3.3.2 Results concerning exclusion

▷ Introduce S

▷ Prove $\mathsf{EXCL_1s} : \{\forall A, B\}\ \{\Pi H : A \subseteq B\}\ \{\forall a : A\}\ (A \setminus a) \subseteq (B \setminus (H\ a))$
$= [\lambda A, B]\ [\lambda H : A \subseteq B]\ [\lambda a : A]\ [\lambda x : A \setminus a]\ \mathsf{pair}\ (H\ x.\mathsf{ev.fst})\ x.\mathsf{ev.snd}$

▷ Introduce A : subset S

▷ Prove $\mathsf{EXCL_0} : \{\forall a : A\}\ (A \setminus a) \subseteq A$
$= [\lambda a : A]\ [\lambda x : A \setminus a]\ x.\mathsf{ev.fst}$

▷ Prove $\mathsf{EXCL_2s} : \{\forall a, b \,|\, A\}\ (a = b) \to (A \setminus a) \subseteq (A \setminus b)$
$= [\lambda a, b \,|\, A]\ [\lambda Q : a = b]\ [\lambda x : A \setminus a]$
$\mathsf{pair}\ x.\mathsf{ev.fst}\ ([\lambda X : b = x]\ x.\mathsf{ev.snd}\ (Q.\mathsf{tran}\ X))$

▷ Prove $\mathsf{EXCL_2} : \{\forall a, b \,|\, A\}\ (a = b) \to (A \setminus a) = (A \setminus b)$
$= [\lambda a, b \,|\, A]\ [\lambda Q : a = b]\ \mathsf{pair}\ (\mathsf{EXCL_2s}\ Q)\ (\mathsf{EXCL_2s}\ Q.\mathsf{symm})$

▷ Discharge A, S

B.3.3.3 Squeezing an isomorphism

▷ Introduce S, T; A and B | subset T

▷ Introduce $f \mid \mathsf{map}\ A\ B$; $f' \mid \mathsf{map}\ B\ A$ and suppose $H : (f' \circ f) = \mathsf{identity}$

▷ Introduce $a \mid A$; $b \mid B$ and suppose $Q : (f\ a) = b$

▷ Introduce $x : A \setminus a$

▷ Suppose $X : b = (f\ x)$

▷ Prove subresult $\mathsf{Q}_1 : ((\mathsf{identity})._{\mathbf{1}}\ a) = (((f' \circ f))._{\mathbf{1}}\ a)$
$= (H\ a).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_2 : (f' \circ f\ a) = (f'\ (f\ a))$
$= \mathsf{rewrite\_compose}\ f'\ f\ a$

▷ Prove subresult $\mathsf{Q}_{3\text{-}1} : (f\ a) = (f._{\mathbf{1}}\ x)$
$= Q.\mathsf{tran}\ X$

▷ Prove subresult $\mathsf{Q}_3 : (f'\ (f\ a)) = (f'\ (f._{\mathbf{1}}\ x))$
$= f'.\mathsf{resp}\ \mathsf{Q}_{3\text{-}1}$

▷ Prove subresult $\mathsf{Q}_4 : (f'\ (f\ x)) = (f' \circ f\ x)$
$= (\mathsf{rewrite\_compose}\ f'\ f\ x).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_5 : (((f' \circ f))._{\mathbf{1}}\ x) = ((\mathsf{identity})._{\mathbf{1}}\ x)$
$= H\ x$

▷ Prove subresult $\mathsf{exclusion} : \mathsf{false}$
$= x.\mathsf{ev.snd}\ ((\mathsf{Q}_1.\mathsf{tran}\ \mathsf{Q}_2).\mathsf{tran}\ (\mathsf{Q}_3.\mathsf{tran}\ (\mathsf{Q}_4.\mathsf{tran}\ \mathsf{Q}_5)))$

▷ Discharge X

▷ Prove $\mathsf{squeeze\_forms\_exclusion} : ((f\ x) \in B) \wedge ((b = (f\ x)) \to \mathsf{false})$
$= \mathsf{pair}\ (f\ x).\mathsf{ev}\ \mathsf{exclusion}$

▷ Define $\mathsf{squeeze\_fun} = (f\ x, \mathsf{squeeze\_forms\_exclusion} : B \setminus b) : B \setminus b$

▷ Discharge x

▷ Prove $\mathsf{squeeze\_forms\_map}$
$: \{\forall x, y \mid A \setminus a\}\ (x = y) \to (\mathsf{squeeze\_fun}\ x) = (\mathsf{squeeze\_fun}\ y)$
$= [\lambda x, y \mid A \setminus a]\ [\lambda Q : x = y]\ f.\mathsf{resp}|x|y\ Q$

▷ Define $\mathsf{squeeze} = (\mathsf{squeeze\_fun}, \mathsf{squeeze\_forms\_map} : \mathsf{map}\ (A \setminus a)\ (B \setminus b))$
$: \mathsf{map}\ (A \setminus a)\ (B \setminus b)$

▷ Prove $\mathsf{SQUEEZE}_0 : \{\forall x : A \setminus a\}\ \mathsf{equal\_in}\ T\ (\mathsf{squeeze}\ x)\ (f\ x)$
$= [\lambda x : A \setminus a]\ \mathsf{refl}|T\ (f\ x)$

▷ Discharge $\mathsf{exclusion}, \mathsf{Q}_5, \mathsf{Q}_4, \mathsf{Q}_3, \mathsf{Q}_{3\text{-}1}, \mathsf{Q}_2, \mathsf{Q}_1, Q, b, a, H, f', f$

▷ Discharge but keep B, A, T, S

▷ Introduce ϕ : iso A B; let $\phi' = \phi^{-1}$: iso B A and introduce $a : A$

▷ Prove subresult Q : $(\phi\ a) = (\phi\ a)$
= refl $(\phi\ a)$

▷ Prove subresult P : $(\phi^{-1} \circ \phi) =$ identity
= inverse_compose_iso ϕ

▷ Let $\psi =$ squeeze P Q : map $(A \setminus a)$ $(B \setminus (\phi\ a))$

▷ Prove subresult Q'$_1$: $(\phi^{-1}\ (\phi\ a)) = (\phi^{-1} \circ \phi\ a)$
= (rewrite_compose ϕ^{-1} ϕ a).symm

▷ Prove subresult Q'$_2$: $(((\phi^{-1} \circ \phi))._{\mathbf{1}}\ a) = ((\text{identity})._{\mathbf{1}}\ a)$
= P a

▷ Prove subresult Q' : $(\phi^{-1}\ (\phi\ a)) = a$
= Q'$_1$.tran Q'$_2$

▷ Prove subresult P' : $(\phi \circ \phi^{-1}) =$ identity
= iso_compose_inverse ϕ

▷ Let $\psi' =$ squeeze P' Q' : map $(B \setminus (\phi\ a))$ $(A \setminus a)$

▷ Prove subresult Q$_0$: $\{\forall x : A \setminus a\}$ equal_in B $(\psi\ x)$ $(\phi\ x)$
= $[\lambda x : A \setminus a]$ SQUEEZE$_0$ P Q x

▷ Prove subresult Q$_0$' : $\{\forall y : B \setminus (\phi\ a)\}$ equal_in A $(\psi'\ y)$ $(\phi'\ y)$
= $[\lambda y : B \setminus (\phi\ a)]$ SQUEEZE$_0$ P' Q' y

▷ Introduce $x : A \setminus a$

▷ Prove subresult Q$_1$: $(\psi' \circ \psi\ x) = (\psi'\ (\psi\ x))$
= rewrite_compose ψ' ψ x

▷ Prove subresult Q$_2$: equal_in A $(\psi'\ (\psi\ x))$ $(\phi'\ (\psi\ x))$
= Q$_0$' $(\psi\ x)$

▷ Prove subresult Q$_3$: $(\phi'\ (\psi\ x)) = (\phi'\ (\phi\ x))$
= ϕ'.resp (Q$_0$ x)

▷ Prove subresult Q$_4$: $(\phi'\ (\phi\ x)) = (\phi' \circ \phi\ x)$
= (rewrite_compose ϕ' ϕ x).symm

▷ Prove subresult Q$_5$: $(((\phi^{-1} \circ \phi))._{\mathbf{1}}\ x) = ((\text{identity})._{\mathbf{1}}\ x)$
= P x

▷ Prove subresult P$_1$: $((\psi' \circ \psi\ x))._{\mathbf{1}} = (((\text{identity})._{\mathbf{1}}\ x))._{\mathbf{1}}$
= (Q$_1$.(tran|S) Q$_2$).tran (Q$_3$.tran (Q$_4$.tran Q$_5$))

▷ Introduce $y : B \setminus (\phi\ a)$

▷ Prove subresult Q_6 **:** $(\psi \circ \psi'\ y) = (\psi\ (\psi'\ y))$
= rewrite_compose $\psi\ \psi'\ y$

▷ Prove subresult Q_7 **:** equal_in $B\ (\psi\ (\psi'\ y))\ (\phi\ (\psi'\ y))$
= $\mathsf{Q}_0\ (\psi'\ y)$

▷ Prove subresult Q_8 **:** $(\phi\ (\psi'\ y)) = (\phi\ (\phi'\ y))$
= ϕ.resp (Q_0' y)

▷ Prove subresult Q_9 **:** $(\phi\ (\phi'\ y)) = (\phi \circ \phi'\ y)$
= (rewrite_compose $\phi\ \phi'\ y$).symm

▷ Prove subresult Q_{10} **:** $(((\phi \circ \phi^{-1}))_{\cdot\mathbf{1}}\ y) = ((\mathsf{identity})_{\cdot\mathbf{1}}\ y)$
= P' y

▷ Prove subresult P_2 **:** $((\psi \circ \psi'\ y))_{\cdot\mathbf{1}} = (((\mathsf{identity})_{\cdot\mathbf{1}}\ y))_{\cdot\mathbf{1}}$
= (Q_6.(tran|T) Q_7).tran (Q_8.tran (Q_9.tran Q_{10}))

▷ Discharge y, x

▷ Prove squeeze_forms_iso **:** $\psi \in (\mathsf{iso}\ (A \setminus a)\ (B \setminus (\phi\ a)))$
= (ψ', pair P_1 P_2 : $\psi \in (\mathsf{iso}\ (A \setminus a)\ (B \setminus (\phi\ a)))$)

▷ Define squeeze_iso = $(\psi, \mathsf{squeeze\_forms\_iso} : \mathsf{iso}\ (A \setminus a)\ (B \setminus (\phi\ a)))$
: iso $(A \setminus a)\ (B \setminus (\phi\ a))$

▷ Discharge $\mathsf{P}_2, \mathsf{Q}_{10}, \mathsf{Q}_9, \mathsf{Q}_8, \mathsf{Q}_7, \mathsf{Q}_6, \mathsf{P}_1, \mathsf{Q}_5, \mathsf{Q}_4, \mathsf{Q}_3, \mathsf{Q}_2, \mathsf{Q}_1, \mathsf{Q}_0$', $\mathsf{Q}_0, \psi', \mathsf{P}', \mathsf{Q}', \mathsf{Q}'_2$, $\mathsf{Q}'_1, \psi, \mathsf{P}, \mathsf{Q}, a, \phi', \phi, B, A, T, S$

B.3.3.4 Stretching an isomorphism

▷ Introduce S, T; A; B | subset T; a **:** A; b **:** B; suppose is_eq **:** A.is_discrete and introduce f **:** map $(A \setminus a)\ (B \setminus b)$

▷ Introduce x **:** A

▷ Let $\mathsf{stretch}_{\_1} = [\lambda\_ : a = x]\ b$ **:** $(a = x) \to B$

▷ Let stretch_exile = $[\lambda Q : a \neq x]\ (x.\mathsf{rep}, \mathsf{pair}\ x.\mathsf{ev}\ Q : A \setminus a)$ **:** $(a \neq x) \to A \setminus a$

▷ Let $\mathsf{stretch}_{\_2} = [\lambda Q : a \neq x]\ f\ (\mathsf{stretch\_exile}\ Q)$ **:** $(a \neq x) \to B$

▷ Discharge x

▷ Define stretch_fun = $[\lambda x : A]$ case $(a.is\_eq\ x)\ (\mathsf{stretch}_{\_1}\ x)\ (\mathsf{stretch}_{\_2}\ x)$
: $A \to B$

▷ Prove stretch_forms_map
: $\{\forall x_1, x_2 \mid A\}\ (x_1 = x_2) \to (\mathsf{stretch\_fun}\ x_1) = (\mathsf{stretch\_fun}\ x_2)$
= $[\lambda x_1, x_2 \mid A]\ [\lambda Q : x_1 = x_2]$ case_elim
($[\lambda X : (a = x_1).\mathsf{or\_not}]$ (case X x_1.stretch_1 x_1.stretch_2) = (stretch_fun x_2))
($a.is\_eq\ x_1$) ($[\lambda Q_1 : a = x_1]$ case_elim
($[\lambda Y : (a = x_2).\mathsf{or\_not}]$ (stretch_1 x_1 Q_1) = (case Y x_2.stretch_1 x_2.stretch_2))
($a.is\_eq\ x_2$) ($[\lambda\_ : a = x_2]$ refl b)
($[\lambda Q_2 : a \neq x_2]$ ex_falso $(b = (\mathsf{stretch\_2}\ x_2\ Q_2))$ $(Q_2\ (Q_1.\mathsf{tran}\ Q))$))
($[\lambda Q_1 : a \neq x_1]$ case_elim
($[\lambda Y : (a = x_2).\mathsf{or\_not}]$ (stretch_2 x_1 Q_1) = (case Y x_2.stretch_1 x_2.stretch_2))
($a.is\_eq\ x_2$)
($[\lambda Q_2 : a = x_2]$ ex_falso $((\mathsf{stretch\_2}\ x_1\ Q_1) = b)$ $(Q_1\ (Q_2.\mathsf{tran}\ Q.\mathsf{symm}))$)
($[\lambda Q_2 : a \neq x_2]$ f.resp|(stretch_exile x_1 Q_1)|(stretch_exile x_2 Q_2) Q))

▷ Define stretch = (stretch_fun, stretch_forms_map : map A B) : map A B

▷ Construct by refinement $\mathsf{STRETCH}_1$: (stretch a) = b
(refl, stretch_2, stretch_1, =, ¬, inr, case, ex_falso, inl, or_not, case_elim)

▷ Discharge $f, is\_eq, b, a, B, A, T, S$

▷ Introduce S, T; A; B | subset T; a : A; b : B; suppose is_eqA : A.is_discrete;
is_eqB : B.is_discrete; introduce f : map $(A \setminus a)$ $(B \setminus b)$; g
: map $(B \setminus b)$ $(A \setminus a)$ and suppose H : $(f \circ g)$ = identity

▷ Let f' = stretch a b is_eqA f : map A B and g' = stretch b a is_eqB g
: map B A

▷ Let Qfa = $\mathsf{STRETCH}_1$ a b is_eqA f : (f' a) = b

▷ Construct by refinement $\mathsf{STRETCH}_2$: (f' ∘ g') = identity
(stretch_exile, stretch_2, refl, \, resp, identity, is_map, ∘, tran, =, ¬, ev, symm,
eq_closed, everything_except, ∈, snd, stretch_1, inl, case, ex_falso, inr, or_not,
case_elim, Qfa, f', subset_forms_set, set, g', rewrite_compose)

▷ Discharge but keep Qfa, g', f', $H, g, f, is\_eqB, is\_eqA, b, a, B, A, T, S$

▷ Discharge Qfa, g', f', H, g, f

▷ Introduce ψ : iso $(A \setminus a)$ $(B \setminus b)$ and let $\psi' = \psi^{-1}$: iso $(B \setminus b)$ $(A \setminus a)$

▷ Let ϕ = stretch a b is_eqA ψ : map A B and ϕ' = stretch b a is_eqB ψ'
: map B A

▷ Prove subresult P_1 : $(\phi' \circ \phi)$ = identity
= $\mathsf{STRETCH}_2$ b a is_eqB is_eqA ψ' ψ (inverse_compose_iso ψ)

▷ Prove subresult P_2 : $(\phi \circ \phi') = \mathsf{identity}$
$= \mathsf{STRETCH}_2\ a\ b\ is\_eqA\ is\_eqB\ \psi\ \psi'\ (\mathsf{iso\_compose\_inverse}\ \psi)$

▷ Prove $\mathsf{stretch\_forms\_iso}$: $\phi \in (\mathsf{iso}\ A\ B)$
$= (\phi', \mathsf{pair}\ \mathsf{P}_1\ \mathsf{P}_2 : \phi \in (\mathsf{iso}\ A\ B))$

▷ Define $\mathsf{stretch\_iso} = (\mathsf{stretch}\ a\ b\ is\_eqA\ \psi, \mathsf{stretch\_forms\_iso} : \mathsf{iso}\ A\ B)$
: $\mathsf{iso}\ A\ B$

▷ Discharge $\mathsf{P}_2, \mathsf{P}_1, \phi', \phi, \psi', \psi, is\_eqB, is\_eqA, b, a, B, A, T, S, \mathsf{stretch}_{-2}$, $\mathsf{stretch\_exile}, \mathsf{stretch}_{-1}$

B.3.3.5 Permuting a discrete set by swapping a pair of elements

▷ Introduce S and is_eq : $S.\mathsf{is\_discrete}$

▷ Introduce a, b : S

▷ Define $\mathsf{permute\_fun} = [\lambda x : S]\ \mathsf{if}\ (x.is\_eq\ a)\ b\ (\mathsf{if}\ (x.is\_eq\ b)\ a\ x)$: $S \to S$

▷ Introduce $x_1, x_2 \mid S$ and Q : $x_1 = x_2$

▷ Suppose H : $x_2 = a$

▷ Prove subresult H_1 : $x_1 = a$
$= Q.\mathsf{tran}\ H$

▷ Prove subresult Q_{1_1} : $(\mathsf{permute\_fun}\ x_1) = b$
$= \mathsf{IF}_1\ b\ (\mathsf{if}\ (x_1.is\_eq\ b)\ a\ x_1)\ (x_1.is\_eq\ a)\ \mathsf{H}_1$

▷ Prove subresult Q_{1_2} : $(\mathsf{permute\_fun}\ x_2) = b$
$= \mathsf{IF}_1\ b\ (\mathsf{if}\ (x_2.is\_eq\ b)\ a\ x_2)\ (x_2.is\_eq\ a)\ H$

▷ Prove subresult C_1 : $(\mathsf{permute\_fun}\ x_1) = (\mathsf{permute\_fun}\ x_2)$
$= \mathsf{Q}_{1\_1}.\mathsf{tran}\ \mathsf{Q}_{1\_2}.\mathsf{symm}$

▷ Discharge H

▷ Suppose H : $x_2 \neq a$

▷ Prove subresult H_2 : $x_1 \neq a$
$= [\lambda contra : x_1 = a]\ H\ (Q.\mathsf{symm.tran}\ contra)$

▷ Prove subresult Q_{2_1} : $(\mathsf{permute\_fun}\ x_1) = (\mathsf{if}\ (x_1.is\_eq\ b)\ a\ x_1)$
$= \mathsf{IF}_0\ b\ (\mathsf{if}\ (x_1.is\_eq\ b)\ a\ x_1)\ (x_1.is\_eq\ a)\ \mathsf{H}_2$

▷ Prove subresult Q_{2_2} : $(\mathsf{permute\_fun}\ x_2) = (\mathsf{if}\ (x_2.is\_eq\ b)\ a\ x_2)$
$= \mathsf{IF}_0\ b\ (\mathsf{if}\ (x_2.is\_eq\ b)\ a\ x_2)\ (x_2.is\_eq\ a)\ H$

▷ Suppose H' : $x_2 = b$

▷ Prove subresult $\mathsf{H_1'}$: $x_1 = b$
= Q.tran H'

▷ Prove subresult $\mathsf{Q_{2\text{-}1\text{-}1}}$: (permute_fun x_1) = a
= $\mathsf{Q_{2\text{-}1}}$.tran ($\mathsf{IF_1}$ a x_1 ($x_1.is\_eq$ b) $\mathsf{H_1'}$)

▷ Prove subresult $\mathsf{Q_{2\text{-}1\text{-}2}}$: (permute_fun x_2) = a
= $\mathsf{Q_{2\text{-}2}}$.tran ($\mathsf{IF_1}$ a x_2 ($x_2.is\_eq$ b) H')

▷ Prove subresult $\mathsf{C_{2\text{-}1}}$: (permute_fun x_1) = (permute_fun x_2)
= $\mathsf{Q_{2\text{-}1\text{-}1}}$.tran $\mathsf{Q_{2\text{-}1\text{-}2}}$.symm

▷ Discharge H'

▷ Suppose H' : $x_2 \neq b$

▷ Prove subresult $\mathsf{H_2'}$: $x_1 \neq b$
= [$\lambda contra : x_1 = b$] H' (Q.symm.tran $contra$)

▷ Prove subresult $\mathsf{Q_{2\text{-}2\text{-}1}}$: (permute_fun x_1) = x_1
= $\mathsf{Q_{2\text{-}1}}$.tran ($\mathsf{IF_0}$ a x_1 ($x_1.is\_eq$ b) $\mathsf{H_2'}$)

▷ Prove subresult $\mathsf{Q_{2\text{-}2\text{-}2}}$: (permute_fun x_2) = x_2
= $\mathsf{Q_{2\text{-}2}}$.tran ($\mathsf{IF_0}$ a x_2 ($x_2.is\_eq$ b) H')

▷ Prove subresult $\mathsf{C_{2\text{-}2}}$: (permute_fun x_1) = (permute_fun x_2)
= ($\mathsf{Q_{2\text{-}2\text{-}1}}$.tran Q).tran $\mathsf{Q_{2\text{-}2\text{-}2}}$.symm

▷ Discharge H'

▷ Prove subresult $\mathsf{C_2}$: (permute_fun x_1) = (permute_fun x_2)
= case ($x_2.is\_eq$ b) $\mathsf{C_{2\text{-}1}}$ $\mathsf{C_{2\text{-}2}}$

▷ Discharge H

▷ Prove permute_forms_map : (permute_fun x_1) = (permute_fun x_2)
= case ($x_2.is\_eq$ a) $\mathsf{C_1}$ $\mathsf{C_2}$

▷ Discharge $\mathsf{C_2}, \mathsf{C_{2\text{-}2}}, \mathsf{Q_{2\text{-}2\text{-}2}}, \mathsf{Q_{2\text{-}2\text{-}1}}, \mathsf{H_2'}, \mathsf{C_{2\text{-}1}}, \mathsf{Q_{2\text{-}1\text{-}2}}, \mathsf{Q_{2\text{-}1\text{-}1}}, \mathsf{H_1'}, \mathsf{Q_{2\text{-}2}}, \mathsf{Q_{2\text{-}1}}, \mathsf{H_2}, \mathsf{C_1}, \mathsf{Q_{1\text{-}2}}, \mathsf{Q_{1\text{-}1}}, \mathsf{H_1}, Q, x_2, x_1$

▷ Define permute_map = (permute_fun, permute_forms_map : map S S) : map S S

▷ Introduce x : S

▷ Prove $\mathsf{PERMUTE_0}$: $(x \neq a) \to (x \neq b) \to$ (permute_map x) = x
= [$\lambda H_1 : x \neq a$] [$\lambda H_2 : x \neq b$]
($\mathsf{IF_0}$ b (if ($x.is\_eq$ b) a x) ($x.is\_eq$ a) H_1).tran ($\mathsf{IF_0}$ a x ($x.is\_eq$ b) H_2)

▷ Prove $\mathsf{PERMUTE}_1 : (x = a) \to (\mathsf{permute\_map}\ x) = b$
$= [\lambda H : x = a]\ \mathsf{IF}_1\ b\ (\mathsf{if}\ (x.is\_eq\ b)\ a\ x)\ (x.is\_eq\ a)\ H$

▷ Suppose $H : x = b$

▷ Prove subresult $\mathsf{C}_1 : (x = a) \to (\mathsf{permute\_map}\ x) = a$
$= [\lambda H' : x = a]\ (\mathsf{PERMUTE}_1\ H').\mathsf{tran}\ (H.\mathsf{symm.tran}\ H')$

▷ Prove subresult $\mathsf{C}_2 : (x \neq a) \to (\mathsf{permute\_map}\ x) = a$
$= [\lambda H' : x \neq a]$
$(\mathsf{IF}_0\ b\ (\mathsf{if}\ (x.is\_eq\ b)\ a\ x)\ (x.is\_eq\ a)\ H').\mathsf{tran}\ (\mathsf{IF}_1\ a\ x\ (x.is\_eq\ b)\ H)$

▷ Prove $\mathsf{PERMUTE}_2 : (\mathsf{permute\_map}\ x) = a$
$= \mathsf{case}\ (x.is\_eq\ a)\ \mathsf{C}_1\ \mathsf{C}_2$

▷ Discharge $\mathsf{C}_2, \mathsf{C}_1, H$

▷ Discharge but keep x

▷ Suppose $H : x = a$

▷ Prove subresult $\mathsf{Q}_{1\text{-}1} : (\mathsf{permute\_map}\ (\mathsf{permute\_map}\ x)) = a$
$= \mathsf{PERMUTE}_2\ (\mathsf{permute\_map}\ x)\ (\mathsf{PERMUTE}_1\ x\ H)$

▷ Prove subresult $\mathsf{Q}_{1\text{-}2} : a = x$
$= H.\mathsf{symm}$

▷ Prove subresult $\mathsf{C}_1 : (\mathsf{permute\_map}\ (\mathsf{permute\_map}\ x)) = x$
$= \mathsf{Q}_{1\text{-}1}.\mathsf{tran}\ \mathsf{Q}_{1\text{-}2}$

▷ Discharge H

▷ Suppose $H_1 : x \neq a$

▷ Suppose $H : x = b$

▷ Prove subresult $\mathsf{Q}_{2\text{-}1\text{-}1} : (\mathsf{permute\_map}\ (\mathsf{permute\_map}\ x)) = b$
$= \mathsf{PERMUTE}_1\ (\mathsf{permute\_map}\ x)\ (\mathsf{PERMUTE}_2\ x\ H)$

▷ Prove subresult $\mathsf{Q}_{2\text{-}1\text{-}2} : b = x$
$= H.\mathsf{symm}$

▷ Prove subresult $\mathsf{C}_{2\text{-}1} : (\mathsf{permute\_map}\ (\mathsf{permute\_map}\ x)) = x$
$= \mathsf{Q}_{2\text{-}1\text{-}1}.\mathsf{tran}\ \mathsf{Q}_{2\text{-}1\text{-}2}$

▷ Discharge H

▷ Suppose $H_2 : x \neq b$

▷ Prove subresult $\mathsf{Q}_{2\text{-}2\text{-}1} : (\mathsf{permute\_map}\ x) = x$
$= \mathsf{PERMUTE}_0\ x\ H_1\ H_2$

▷ Prove subresult $\mathsf{Q}_{2\text{-}2\text{-}2}$ **:** $(\mathsf{permute\_map}\ (\mathsf{permute\_map}\ x)) = (\mathsf{permute\_map}\ x)$
$= \mathsf{permute\_map.resp}\ \mathsf{Q}_{2\text{-}2\text{-}1}$

▷ Prove subresult $\mathsf{C}_{2\text{-}2}$ **:** $(\mathsf{permute\_map}\ (\mathsf{permute\_map}\ x)) = x$
$= \mathsf{Q}_{2\text{-}2\text{-}2}.\mathsf{tran}\ \mathsf{Q}_{2\text{-}2\text{-}1}$

▷ Discharge H_2

▷ Prove subresult C_2 **:** $(\mathsf{permute\_map}\ (\mathsf{permute\_map}\ x)) = x$
$= \mathsf{case}\ (x.is\_eq\ b)\ \mathsf{C}_{2\text{-}1}\ \mathsf{C}_{2\text{-}2}$

▷ Discharge H_1

▷ Prove subresult $\mathsf{permute\_is\_idempotent}$ **:** $(\mathsf{permute\_map}\ (\mathsf{permute\_map}\ x)) = x$
$= \mathsf{case}\ (x.is\_eq\ a)\ \mathsf{C}_1\ \mathsf{C}_2$

▷ Discharge x

▷ Prove $\mathsf{permute\_forms\_iso}$ **:** $\mathsf{permute\_map} \in (\mathsf{iso}\ S\ S)$
$= (\mathsf{permute\_map}, \mathsf{pair}\ \mathsf{permute\_is\_idempotent}\ \mathsf{permute\_is\_idempotent}$:
$\mathsf{permute\_map} \in (\mathsf{iso}\ S\ S))$

▷ Define $\mathsf{permute} = (\mathsf{permute\_map}, \mathsf{permute\_forms\_iso} : \mathsf{iso}\ S\ S)$ **:** $\mathsf{iso}\ S\ S$

▷ Discharge $\mathsf{permute\_is\_idempotent}, \mathsf{C}_2, \mathsf{C}_{2\text{-}2}, \mathsf{Q}_{2\text{-}2\text{-}2}, \mathsf{Q}_{2\text{-}2\text{-}1}, \mathsf{C}_{2\text{-}1}, \mathsf{Q}_{2\text{-}1\text{-}2}, \mathsf{Q}_{2\text{-}1\text{-}1}, \mathsf{C}_1, \mathsf{Q}_{1\text{-}2}, \mathsf{Q}_{1\text{-}1}, b, a, is\_eq, S$

B.4 Finiteness

B.4.1 The natural numbers

B.4.1.1 The set of natural numbers

▷ Globally declare $\mathsf{nat} : \mathsf{Type}_0$; $\mathsf{zero} : \mathsf{nat}$; $\mathsf{succ} : \mathsf{nat} \to \mathsf{nat}$; nat_elim

$: \{\Pi C\_nat : \mathsf{nat} \to \mathsf{Type}_1\}\ (C\_nat\ \mathsf{zero}) \to$

$(\{\Pi x_1 : \mathsf{nat}\}\ (C\_nat\ x_1) \to C\_nat\ (\mathsf{succ}\ x_1)) \to \{\Pi z : \mathsf{nat}\}\ C\_nat\ z$ and

define nat_double_elim

$= [\lambda T\_nat : \mathsf{nat} \to \mathsf{nat} \to \mathsf{Type}_1]\ [\lambda zero\_zero\_case : T\_nat\ \mathsf{zero}\ \mathsf{zero}]$

$[\lambda zero\_succ\_case : \{\Pi x'_1 : \mathsf{nat}\}\ (T\_nat\ \mathsf{zero}\ x'_1) \to T\_nat\ \mathsf{zero}\ (\mathsf{succ}\ x'_1)]$

$[\lambda succ\_zero\_case : \{\Pi x_1 : \mathsf{nat}\}\ (\{\Pi z : \mathsf{nat}\}\ T\_nat\ x_1\ z) \to$

$T\_nat\ (\mathsf{succ}\ x_1)\ \mathsf{zero}]\ [\lambda succ\_succ\_case : \{\Pi x_1 : \mathsf{nat}\}$

$(\{\Pi z : \mathsf{nat}\}\ T\_nat\ x_1\ z) \to \{\Pi x'_1 : \mathsf{nat}\}\ (T\_nat\ (\mathsf{succ}\ x_1)\ x'_1) \to$

$T\_nat\ (\mathsf{succ}\ x_1)\ (\mathsf{succ}\ x'_1)]$ nat_elim $([\lambda z : \mathsf{nat}]\ \{\Pi z' : \mathsf{nat}\}\ T\_nat\ z\ z')$

$(\mathsf{nat\_elim}\ ([\lambda z : \mathsf{nat}]\ T\_nat\ \mathsf{zero}\ z)\ zero\_zero\_case\ zero\_succ\_case)$

$([\lambda x_1 : \mathsf{nat}]\ [\lambda x_1\_f : \{\Pi z : \mathsf{nat}\}\ T\_nat\ x_1\ z]$

$\mathsf{nat\_elim}\ ([\lambda z : \mathsf{nat}]\ T\_nat\ (\mathsf{succ}\ x_1)\ z)\ (succ\_zero\_case\ x_1\ x_1\_f)$

$(succ\_succ\_case\ x_1\ x_1\_f))$

$: \{\Pi T\_nat : \mathsf{nat} \to \mathsf{nat} \to \mathsf{Type}_1\}\ (T\_nat\ \mathsf{zero}\ \mathsf{zero}) \to$

$(\{\Pi x'_1 : \mathsf{nat}\}\ (T\_nat\ \mathsf{zero}\ x'_1) \to T\_nat\ \mathsf{zero}\ (\mathsf{succ}\ x'_1)) \to$

$(\{\Pi x_1 : \mathsf{nat}\}\ (\{\Pi z : \mathsf{nat}\}\ T\_nat\ x_1\ z) \to T\_nat\ (\mathsf{succ}\ x_1)\ \mathsf{zero}) \to$

$(\{\Pi x_1 : \mathsf{nat}\}\ (\{\Pi z : \mathsf{nat}\}\ T\_nat\ x_1\ z) \to \{\Pi x'_1 : \mathsf{nat}\}\ (T\_nat\ (\mathsf{succ}\ x_1)\ x'_1) \to$

$T\_nat\ (\mathsf{succ}\ x_1)\ (\mathsf{succ}\ x'_1)) \to \{\Pi z, z' : \mathsf{nat}\}\ T\_nat\ z\ z'$

▷ Reductions: $[\lambda C\_nat : \mathsf{nat} \to \mathsf{Type}_1]\ [\lambda f\_zero : C\_nat\ \mathsf{zero}]$

$[\lambda f\_succ : \{\Pi x_1 : \mathsf{nat}\}\ (C\_nat\ x_1) \to C\_nat\ (\mathsf{succ}\ x_1)]\ [\lambda x_1 : \mathsf{nat}]$

$\mathsf{nat\_elim}\ C\_nat\ f\_zero\ f\_succ\ \mathsf{zero} \Longrightarrow f\_zero$

$\|\ \mathsf{nat\_elim}\ C\_nat\ f\_zero\ f\_succ\ (\mathsf{succ}\ x_1) \Longrightarrow$

$f\_succ\ x_1\ (\mathsf{nat\_elim}\ C\_nat\ f\_zero\ f\_succ\ x_1)$

▷ Define $\mathsf{nat\_rec} = [\lambda C \mid \mathsf{Type}_1]\ \mathsf{nat\_elim}\ ([\lambda\_ : \mathsf{nat}]\ C)$

$: \{\Pi C \mid \mathsf{Type}_1\}\ C \to (\mathsf{nat} \to C \to C) \to \mathsf{nat} \to C$

▷ Construct by refinement $\mathsf{nat\_eq} : \mathsf{rel\ nat}$

(prop, nat, false, nat_rec, true)

▷ Construct by refinement nat_eq_resp

$: \{\forall x, y \mid \mathsf{nat}\}\ (x.\mathsf{nat\_eq}\ y) \to \{\forall p : \mathsf{nat} \to \mathsf{prop}\}\ (p\ x) \to p\ y$

(succ, nat, prop, nat_eq, zero, ex_falso, nat_double_elim)

▷ Construct by refinement $\mathsf{nat\_forms\_set} : \mathsf{set\_axioms\ nat\_eq}$

(nat_eq, nat, τ, nat_elim, nat_eq_resp, is_tran, is_symm, is_refl, pair_3)

▷ Define $\mathbb{N} = (\mathsf{nat}, \mathsf{nat\_eq}, \mathsf{nat\_forms\_set} : \mathsf{set}) : \mathsf{set}$

▷ Unless otherwise specified, by default $m, n : \mathbb{N}$

▷ Define $0 = \mathsf{zero} : \mathbb{N}$

▷ Construct by refinement $\mathsf{succ\_forms\_map} : ([\lambda n]\ \mathsf{succ}\ n).\mathsf{is\_map}$

($\mathbb{N}$, =, nat_forms_set, nat_eq, nat, is_tran, is_symm, is_refl, fst_3, nat_eq_resp)

▷ Allow $+1$ to be written postfix

▷ Define $+1 = ([\lambda n]\ \mathsf{succ}\ n, \mathsf{succ\_forms\_map} : \mathsf{map}\ \mathbb{N}\ \mathbb{N}) : \mathsf{map}\ \mathbb{N}\ \mathbb{N}$

▷ Define $1 = 0{+}1 : \mathbb{N}$

▷ Define $\mathsf{N\_elim} = \mathsf{nat\_elim} : \{\Pi C\_N : \mathbb{N} \to \mathsf{Type}_1\}\ (C\_N\ 0) \to$

$(\{\Pi n\}\ (C\_N\ n) \to C\_N\ n{+}1) \to \{\Pi n\}\ C\_N\ n$

▷ Define $\mathsf{N\_rec} = \mathsf{nat\_rec} : \{\Pi C \mid \mathsf{Type}_1\}\ C \to (\mathbb{N} \to C \to C) \to \mathbb{N} \to C$

B.4.1.2 Results concerning the natural numbers

▷ Construct by refinement $\mathsf{NAT}_1 : \{\forall n\}\ (0 = n) \vee (\langle \exists m \rangle\ m{+}1 = n)$

($+1$, $\mathbb{N}$, refl, succ, =, 0, inr, $\vee$, nat, τ, zero, true, inl, nat_elim)

B.4.1.3 The standard order on the natural numbers

▷ Allow $<$ to be written infix

▷ Define $< =$ nat_rec (nat_rec false ($[\lambda\_ : \mathsf{nat}]$ $[\lambda\_ : \mathsf{prop}]$ true))

($[\lambda\_ : \mathsf{nat}]$ $[\lambda H : \mathbb{N} \to \mathsf{prop}]$ nat_rec false ($[\lambda m : \mathsf{nat}]$ $[\lambda\_ : \mathsf{prop}]$ H m)) : rel $\mathbb{N}$

▷ Allow $\leq$ to be written infix

▷ Define $\leq = [\lambda x, y : \mathbb{N}]$ $(x < y) \vee (x = y)$: rel $\mathbb{N}$

B.4.1.4 Results concerning the standard order on the natural numbers

▷ Construct by refinement LESS_0 : $\{\forall n \,|\, \mathbb{N}\}$ $\neg(n < 0)$

(0, succ, $<$, $\neg$, nat, zero, $\mathbb{N}$, nat_elim)

▷ Construct by refinement less_than_tran : $<$.is_tran

($+1$, $<$, nat, zero, $\mathbb{N}$, nat_elim, succ, LESS_0, ex_falso, τ, 0, nat_double_elim)

▷ Construct by refinement less_than_asymm
: $\{\forall m, n \,|\, \mathbb{N}\}$ $(m < n) \to \neg(n < m)$

(succ, $<$, $\neg$, nat, LESS_0, zero, ex_falso, $\mathbb{N}$, nat_double_elim)

▷ Prove less_than_arefl : $\{\forall n\}$ $\neg(n < n)$

$= [\lambda n]$ $[\lambda H : n < n]$ less_than_asymm H H

▷ Construct by refinement LESS_1 : $\{\forall n\}$ $n < n{+}1$

($+1$, $<$, nat, τ, $\mathbb{N}$, nat_elim)

▷ Construct by refinement leq_than_tran : $\leq$.is_tran

($\leq$, nat, $\mathbb{N}$, symm, nat_eq_resp, $=$, $<$, less_than_tran, case, inl)

▷ Construct by refinement leq_than_refl : $\leq$.is_refl

($\mathbb{N}$, refl, $=$, $<$, inr)

▷ Construct by refinement leq_than_asymm
: $\{\forall m, n \,|\, \mathbb{N}\}$ $(m \leq n) \to (n \leq m) \to m = n$

($\mathbb{N}$, $=$, symm, less_than_asymm, ex_falso, $<$, case, $\leq$)

▷ Construct by refinement LEQ_1 : $\{\forall m, n\}$ $(m \leq n) \to m < n{+}1$

(succ, $+1$, $\leq$, $<$, nat, τ, 1, $\mathbb{N}$, zero, symm, nat_eq_resp, $=$, LESS_0, ex_falso, case, nat_elim)

▷ Construct by refinement LEQ_2 : $\{\forall m, n\}$ $(m < n{+}1) \to m \leq n$

(succ, $+1$, $<$, $\leq$, nat, LESS_0, zero, ex_falso, $\mathbb{N}$, nat_elim, τ, 0, $=$, inl, inr)

B.4.2 Finite sets

B.4.2.1 Canonical n-element subsets

▷ Construct by refinement less_than_forms_subset
: $\{\forall n\}$ subset_axioms $([\lambda m]\ m < n)$
$(<, \mathbb{N}, \text{nat\_eq\_resp}, =)$

▷ Define canon_subset
$= [\lambda n]\ ([\lambda m]\ m < n, \text{less\_than\_forms\_subset}\ n : \text{subset}\ \mathbb{N}) : \mathbb{N} \to \text{subset}\ \mathbb{N}$

▷ Define coercion canonical_subset = canon_subset : $\mathbb{N} \to$ subset $\mathbb{N}$

▷ Define $\underline{0} = [\lambda n]\ (0, \tau : n{+}1) : \{\Pi n\}\ n{+}1$

▷ Define last_ $= [\lambda n]\ (n, \mathsf{LESS}_1\ n : n{+}1) : \{\Pi n\}\ n{+}1$

▷ Introduce $n \mid \mathbb{N}$

▷ Unless otherwise specified, by default $i, j, k : n$

▷ Construct by refinement s__forms_map
: $([\lambda i]\ (i.\mathbf{1}{+}1, i.\mathbf{2} : n{+}1)).(\text{is\_map}|n|n{+}1)$
$(+1, \mathbb{N}, \text{resp}, =)$

▷ Allow $+\underline{1}$ to be written postfix

▷ Define $+\underline{1} = ([\lambda i]\ (i.\mathbf{1}{+}1, i.\mathbf{2} : n{+}1), \text{s\_\_forms\_map} : \text{map}\ n\ n{+}1)$
: map $n\ n{+}1$

▷ Allow $\underline{0}+$ to be written prefix

▷ Define $\underline{0}+ = [\lambda i]\ (i.\text{rep}, i.\text{ev.less\_than\_tran}\ (\mathsf{LESS}_1\ n) : n{+}1) : n \to n{+}1$

▷ Discharge n

B.4.2.2 Comparing the sizes of finite subsets

▷ Introduce S; A : subset S; T and B : subset T

▷ Allow $\cong$ to be written infix

▷ Define $\cong$ = el (iso $B\ A$) : prop

▷ Discharge B, T

▷ Define has_finite_size $= [\lambda n] \cong n : \mathbb{N} \to$ prop

▷ Define $\mathsf{is\_finite} = \langle \exists n \rangle$ $\mathsf{has\_finite\_size}$ n : prop

▷ Discharge but keep A, S

▷ Introduce T and B : subset T

▷ Allow $\preccurlyeq$ to be written infix

▷ Define $\preccurlyeq = \langle \exists a, b : \mathbb{N} \rangle \bigwedge_3 (A.\mathsf{has\_finite\_size}\ a)\ (B.\mathsf{has\_finite\_size}\ b)\ (a \leq b)$
: prop

▷ Discharge B, T, A, S

B.4.3 Results concerning finiteness

B.4.3.1 Results concerning canonical n-element subsets

▷ Prove $\mathsf{ex\_O} : \{\Pi C : \mathsf{Type}_0\}\ 0 \to C$
$= [\lambda C : \mathsf{Type}_0]\ [\lambda i : 0]\ \mathsf{ex\_falso}\ C\ (\mathsf{LESS}_0\ i.\mathsf{ev})$

▷ Introduce n

▷ Introduce i : $n{+}1 \setminus (\mathsf{last\_}\ n)$

▷ Prove subresult P_0 : $i \leq n$
$= \mathsf{LEQ}_2\ i\ n\ i.\mathsf{ev.fst}$

▷ Prove subresult C_1 : $(i < n) \to i < n$
$= [\lambda H_1 : i < n]\ H_1$

▷ Prove subresult C_2 : $(\mathsf{equal\_in}\ \mathbb{N}\ i\ n) \to i < n$
$= [\lambda H_2 : \mathsf{equal\_in}\ \mathbb{N}\ i\ n]\ \mathsf{ex\_falso}\ (i < n)\ (i.\mathsf{ev.snd}\ H_2.\mathsf{symm})$

▷ Prove subresult P_1 : $i < n$
$= \mathsf{case}\ \mathsf{P}_0\ \mathsf{C}_1\ \mathsf{C}_2$

▷ Discharge i

▷ Introduce i

▷ Prove subresult $\mathsf{P}_{2\text{-}1}$: $i < n{+}1$
$= i.\mathsf{ev.less\_than\_tran}\ (\mathsf{LESS}_1\ n)$

▷ Suppose X : $\mathsf{equal\_in}\ \mathbb{N}\ (\mathsf{last\_}\ n)\ i$

▷ Prove subresult $\mathsf{P}_{2\text{-}2\text{-}1}$: $i < i$
$= \mathsf{nat\_eq\_resp}\ X\ i < i.\mathsf{ev}$

▷ Prove subresult $\mathsf{P}_{2\text{-}2}$: false
= less_than_arefl i $\mathsf{P}_{2\text{-}2\text{-}1}$

▷ Discharge X

▷ Prove subresult P_2 : $i \in (n{+}1 \setminus (\mathsf{last\_}\ n))$
= pair $\mathsf{P}_{2\text{-}1}$ $\mathsf{P}_{2\text{-}2}$

▷ Discharge i

▷ Prove CANON_1 : $(n{+}1 \setminus (\mathsf{last\_}\ n)) = n$
= pair P_1 P_2

▷ Discharge $\mathsf{P}_2, \mathsf{P}_{2\text{-}2}, \mathsf{P}_{2\text{-}2\text{-}1}, \mathsf{P}_{2\text{-}1}, \mathsf{P}_1, \mathsf{C}_2, \mathsf{C}_1, \mathsf{P}_0, n$

▷ Introduce $n \mid \mathbb{N}$

▷ Construct by refinement CANON_2
: $\{\forall i : n{+}1\}$ (equal_in $n{+}1$ ($\underline{0}$ n) i) $\vee$ ($\langle\exists j\rangle$ equal_in $n{+}1$ $j{+}\underline{1}$ i)
($+1$, $<$, $\mathbb{N}$, symm, nat_eq_resp, canon_subset, $\in$, $+\underline{1}$, equal_in, $\underline{0}$, inr, $=$, 0, inl, NAT_1, $\vee$, case)

▷ Discharge n

▷ Introduce i : 1

▷ Prove subresult C_1 : (equal_in $0{+}1$ ($\underline{0}$ 0) i) $\to 0 = i$
= $[\lambda H_1 : \text{equal\_in } 0{+}1\ (\underline{0}\ 0)\ i]\ H_1$

▷ Prove subresult C_2 : ($\langle\Sigma j : 0\rangle$ equal_in $0{+}1$ $j{+}\underline{1}$ i) $\to 0 = i$
= $[\lambda H_2 : \langle\Sigma j : 0\rangle \text{ equal\_in } 0{+}1\ j{+}\underline{1}\ i]$ ex_O $(0 = i)$ $H_{2\cdot\mathbf{1}}$

▷ Prove CANON_3 : $0 = i$
= case (CANON_2 i) C_1 C_2

▷ Discharge $\mathsf{C}_2, \mathsf{C}_1, i$

B.4.3.2 Results concerning discreteness

▷ Prove N_discrete : $\mathbb{N}$.is_discrete
= nat_double_elim ($[\lambda x, y : \mathbb{N}]$ $(x = y)$.or_not) (or_not_is_true τ)
($[\lambda y : \mathbb{N}]$ [_: $(0 = y)$.or_not] or_not_is_false (ex_falso false))
($[\lambda x : \mathbb{N}]$ [_: $\{\Pi y : \mathbb{N}\}$ $(x = y)$.or_not] or_not_is_false (ex_falso false))
($[\lambda x : \mathbb{N}]$ [ih : $\{\Pi y : \mathbb{N}\}$ $(x = y)$.or_not] [$\lambda y : \mathbb{N}$] [_: $(x{+}1 = y)$.or_not] ih y)

▷ Prove n_discrete : $\{\Pi n\}$ n.is_discrete
= $[\lambda n]$ $[\lambda i, j]$ N_discrete i j

▷ Introduce S, T; suppose H : S.is_discrete and introduce f : split_epi S T

▷ Let $\mathsf{f'} = (f.\mathsf{ev})._{\mathbf{1}} : \mathsf{map}\ T\ S$; introduce $x, y : T$ and suppose H_1 $: (\mathsf{f'}\ x) = (\mathsf{f'}\ y)$

▷ Prove subresult $\mathsf{Q}_1 : x = (f \circ \mathsf{f'}\ x)$
$= ((f.\mathsf{ev})._{\mathbf{2}}\ x).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_2 : (f \circ \mathsf{f'}\ x) = (f\ (\mathsf{f'}\ x))$
$= \mathsf{rewrite\_compose}\ f\ \mathsf{f'}\ x$

▷ Prove subresult $\mathsf{Q}_3 : (f\ (\mathsf{f'}\ x)) = (f\ (\mathsf{f'}\ y))$
$= f.\mathsf{resp}\ H_1$

▷ Prove subresult $\mathsf{Q}_4 : (f\ (\mathsf{f'}\ y)) = (f \circ \mathsf{f'}\ y)$
$= (\mathsf{rewrite\_compose}\ f\ \mathsf{f'}\ y).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_5 : (f \circ \mathsf{f'}\ y) = y$
$= (f.\mathsf{ev})._{\mathbf{2}}\ y$

▷ Prove subresult $\mathsf{C}_1 : x = y$
$= (\mathsf{Q}_1.\mathsf{tran}\ \mathsf{Q}_2).\mathsf{tran}\ (\mathsf{Q}_3.\mathsf{tran}\ (\mathsf{Q}_4.\mathsf{tran}\ \mathsf{Q}_5))$

▷ Discharge H_1

▷ Prove subresult $\mathsf{C}_2 : (x = y) \to (\mathsf{f'}\ x) = (\mathsf{f'}\ y)$
$= [\lambda H_2 : x = y]\ \mathsf{f'}.\mathsf{resp}\ H_2$

▷ Prove subresult $\mathsf{P}_1 : ((\mathsf{f'}\ x) = (\mathsf{f'}\ y)) \leftrightarrow (x = y)$
$= \mathsf{pair}\ \mathsf{C}_1\ \mathsf{C}_2$

▷ Prove subresult $\mathsf{P}_2 : ((\mathsf{f'}\ x) = (\mathsf{f'}\ y)) \vee ((\mathsf{f'}\ x) \neq (\mathsf{f'}\ y))$
$= H\ (\mathsf{f'}\ x)\ (\mathsf{f'}\ y)$

▷ Prove $\mathsf{DISC}_1 : (x = y).\mathsf{or\_not}$
$= \mathsf{DEC}_2\ \mathsf{P}_1\ \mathsf{P}_2$

▷ Discharge $\mathsf{P}_2, \mathsf{P}_1, \mathsf{C}_2, \mathsf{C}_1, \mathsf{Q}_5, \mathsf{Q}_4, \mathsf{Q}_3, \mathsf{Q}_2, \mathsf{Q}_1, y, x, \mathsf{f'}, f, H, T, S$

▷ Introduce S; A and suppose $H : A.\mathsf{is\_finite}$

▷ Let $\mathsf{f} = \mathsf{make}\ ((\mathsf{iso\_eq\_mono\_epi}|H._{\mathbf{1}}|A).\mathsf{fst}\ H._{\mathbf{2}}).\mathsf{snd} : \mathsf{split\_epi}\ H._{\mathbf{1}}\ A$

▷ Prove $\mathsf{DISC}_2 : A.\mathsf{is\_discrete}$
$= \mathsf{DISC}_1\ (\mathsf{n\_discrete}\ H._{\mathbf{1}})\ \mathsf{f}$

▷ Discharge f, H, A, S

B.4.3.3 Results relating exclusion and finite size

▷ Introduce S; A; $n \mid \mathbb{N}$; suppose $a : n{+}1.\mathsf{iso}\ A$ and introduce $x : A$

▷ Let hole_in_n $= a^{-1}\ x$: $n{+}1$

▷ Let swap_iso = permute (n_discrete $n{+}1$) (last_ n) hole_in_n : iso $n{+}1$ $n{+}1$

▷ Let adjusted_a = compose_isos_is_iso a swap_iso : iso $n{+}1$ A

▷ Prove subresult Q_0 : (swap_iso (last_ n)) = hole_in_n
= $\mathsf{PERMUTE}_1$ (n_discrete $n{+}1$) (last_ n) hole_in_n (last_ n) (refl (last_ n))

▷ Prove subresult Q_1 : (adjusted_a (last_ n)) = (a hole_in_n)
= a.resp Q_0

▷ Prove subresult Q_2 : (a hole_in_n) = x
= (rewrite_compose $a\ a^{-1}\ x$).tran (iso_compose_inverse $a\ x$)

▷ Prove subresult squeezed_a : iso ($n{+}1 \setminus$ (last_ n)) ($A \setminus$ (adjusted_a (last_ n)))
= squeeze_iso adjusted_a (last_ n)

▷ Prove subresult Q_3 : ($A \setminus$ (adjusted_a (last_ n))) = ($A \setminus x$)
= EXCL_2 A (Q_1.tran Q_2)

▷ Prove subresult Qn : ($n{+}1 \setminus$ (last_ n)) = n
= CANON_1 n

▷ Prove EXCL_4 : n.iso ($A \setminus x$)
= ISOresp Qn Q_3 squeezed_a

▷ Discharge

Qn, Q_3, squeezed_a, Q_2, Q_1, Q_0, adjusted_a, swap_iso, hole_in_n, x, a, n, A, S

B.4.3.4 Results concerning the sizes of finite subsets

▷ Prove eqsize_refl : $\{\forall S\}$ $\{\forall A : \mathsf{subset}\ S\}$ $A \cong A$
= $[\lambda S]$ $[\lambda A : \mathsf{subset}\ S]$ iso_refl A

▷ Introduce S, S_1, S_2, S_3; A | subset S_1; B | subset S_2 and C | subset S_3

▷ Prove eqsize_symm : $(A \cong B) \to B \cong A$
= $[\lambda H : A \cong B]$ iso_symm H

▷ Prove eqsize_tran : $(A \cong B) \to (B \cong C) \to A \cong C$
= $[\lambda H_1 : A \cong B]$ $[\lambda H_2 : B \cong C]$ iso_tran H_2 H_1

▷ Discharge but keep C, B, A, S_3, S_2, S_1

▷ Prove EQSIZE_0 : $(A \cong B) \to A$.is_finite $\to B$.is_finite
= $[\lambda AqB : A \cong B]$ $[\lambda H : A.\mathsf{is\_finite}]$
($H.\mathbf{1}, AqB$.eqsize_symm.eqsize_tran $H.\mathbf{2}$: B.is_finite)

▷ Let p = $[\lambda m, n]$ $(m \cong n) \to m = n$: $\mathbb{N} \to \mathbb{N} \to$ prop

▷ Prove subresult $\mathsf{Czz} : \mathsf{p}\ 0\ 0$
$= [\lambda\_ : 0 \cong 0]\ \tau$

▷ Prove subresult $\mathsf{Czs} : \{\Pi n\}\ (\mathsf{p}\ 0\ n) \to \mathsf{p}\ 0\ n{+}1$
$= [\lambda n]\ [\lambda\_ : \mathsf{p}\ 0\ n]\ [\lambda X : n{+}1.\mathsf{iso}\ 0]\ \mathsf{ex\_O}\ (0 = n{+}1)\ (X\ (\underline{0}\ n))$

▷ Prove subresult $\mathsf{Cz} : \{\Pi n\}\ \mathsf{p}\ 0\ n$
$= \mathsf{N\_elim}\ ([\lambda n]\ \mathsf{p}\ 0\ n)\ \mathsf{Czz}\ \mathsf{Czs}$

▷ Introduce m and suppose $ih : \{\Pi n\}\ \mathsf{p}\ m\ n$

▷ Prove subresult $\mathsf{Csz} : \mathsf{p}\ m{+}1\ 0$
$= [\lambda X : 0.\mathsf{iso}\ m{+}1]\ \mathsf{ex\_O}\ (m{+}1 = 0)\ (X^{-1}\ (\underline{0}\ m))$

▷ Prove subresult $\mathsf{Css} : \{\Pi n\}\ (\mathsf{p}\ m{+}1\ n) \to \mathsf{p}\ m{+}1\ n{+}1$
$= [\lambda n]\ [\lambda\_ : \mathsf{p}\ m{+}1\ n]\ [\lambda H : n{+}1.\mathsf{iso}\ m{+}1]$
$ih\ n\ (\mathsf{ISO}_2\ n\ (\mathsf{CANON}_1\ m)\ (\mathsf{EXCL}_4\ H\ (\mathsf{last\_}\ m)))$

▷ Prove subresult $\mathsf{Cs} : \{\Pi n\}\ \mathsf{p}\ m{+}1\ n$
$= \mathsf{N\_elim}\ ([\lambda n]\ \mathsf{p}\ m{+}1\ n)\ \mathsf{Csz}\ \mathsf{Css}$

▷ Discharge ih, m

▷ Prove $\mathsf{EQSIZE}_1 : \{\Pi m, n \,|\, \mathbb{N}\}\ (m \cong n) \to m = n$
$= [\lambda m, n \,|\, \mathbb{N}]\ \mathsf{N\_elim}\ ([\lambda m']\ \{\Pi n'\}\ \mathsf{p}\ m'\ n')\ \mathsf{Cz}\ \mathsf{Cs}\ m\ n$

▷ Prove EQSIZE_2
$: \{\Pi m, n \,|\, \mathbb{N}\}\ (A.\mathsf{has\_finite\_size}\ m) \to (A.\mathsf{has\_finite\_size}\ n) \to m = n$
$= [\lambda m, n \,|\, \mathbb{N}]\ [\lambda H_1 : A.\mathsf{has\_finite\_size}\ m]\ [\lambda H_2 : A.\mathsf{has\_finite\_size}\ n]$
$\mathsf{EQSIZE}_1\ (H_1.\mathsf{eqsize\_symm.eqsize\_tran}\ H_2)$

▷ Discharge $\mathsf{Cs}, \mathsf{Css}, \mathsf{Csz}, \mathsf{Cz}, \mathsf{Czs}, \mathsf{Czz}, \mathsf{p}$

▷ Discharge but keep C, B, A, S_3, S_2, S_1

▷ Prove $\mathsf{SMALLER}_0 : (A \preccurlyeq B) \to A.\mathsf{is\_finite}$
$= [\lambda H : A \preccurlyeq B]\ (H.\mathbf{1}, H.\mathbf{2}.\mathsf{fst}_3 : A.\mathsf{is\_finite})$

▷ Construct by refinement $\mathsf{SMALLER}_1 : A.\mathsf{is\_finite} \to (A \cong B) \to A \preccurlyeq B$
(leq_than_refl, eqsize_symm, $\mathbb{N}$, eqsize_tran, $\leq$, has_finite_size, pair_3, $\bigwedge_3$, $\cong$, is_finite)

▷ Construct by refinement $\mathsf{smaller\_tran} : (A \preccurlyeq B) \to (B \preccurlyeq C) \to A \preccurlyeq C$
($\leq$, has_finite_size, thd_3, leq_than_refl, fst_3, snd_3, EQSIZE_2, nat_eq_resp, leq_than_tran, pair_3, $\bigwedge_3$, $\mathbb{N}$, $\preccurlyeq$)

▷ Construct by refinement $\mathsf{smaller\_asymm} : (B \preccurlyeq A) \to (A \preccurlyeq B) \to A \cong B$
($\mathbb{N}$, eqsize_refl, $\cong$, leq_than_refl, $\leq$, has_finite_size, fst_3, snd_3, EQSIZE_2, nat_eq_resp, thd_3, leq_than_tran, leq_than_asymm, eqsize_tran, eqsize_symm, $\preccurlyeq$)

▷ Discharge C, B, A, S_3, S_2, S_1

▷ Let $\mathsf{p}_2 = [\lambda n]\ [\lambda A : \mathsf{subset}\ S]\ \langle \exists m \rangle\ (m.\mathsf{iso}\ A).\mathsf{el} \wedge (m \leq n)$

$: \mathbb{N} \to (\mathsf{subset}\ S) \to \mathsf{prop}$

▷ Let $\mathsf{p}_1 = [\lambda n]\ \{\forall A, B\}\ (A \subseteq B) \to$

$\wedge\ ((n.\mathsf{iso}\ A) \to B.\mathsf{is\_discrete} \to A.\mathsf{is\_decideable\_in}\ B)$

$((n.\mathsf{iso}\ B) \to ((A.\mathsf{is\_decideable\_in}\ B) \to \mathsf{p}_2\ n\ A) \wedge ((n.\mathsf{iso}\ A) \to A = B))$

$: \mathbb{N} \to \mathsf{prop}_1$

▷ Introduce A, B and suppose $H : A \subseteq B$

▷ Let $\mathsf{Cz}_1 = [\lambda a : 0.\mathsf{iso}\ A]\ [\lambda\_ : B.\mathsf{is\_discrete}]\ [\lambda x : B]$

$\mathsf{inr\_is\_true}\ (x \in A)\ ([\lambda X : x \in A]\ \mathsf{ex\_O}\ \mathsf{false}\ (a^{-1}\ (\mathsf{make}\ X)))$

$: (0.\mathsf{iso}\ A) \to B.\mathsf{is\_discrete} \to A.\mathsf{is\_decideable\_in}\ B$

▷ Introduce $b : 0.\mathsf{iso}\ B$

▷ Prove subresult $\mathsf{ex\_A} : \{\Pi C : \mathsf{Type}_0\}\ A \to C$
$= [\lambda C : \mathsf{Type}_0]\ [\lambda x : A]\ \mathsf{ex\_O}\ C\ (b^{-1}\ (H\ x))$

▷ Let $\mathsf{z\_fun} = \mathsf{ex\_O}\ A : 0 \to A$

▷ Let $\mathsf{z'\_fun} = \mathsf{ex\_A}\ 0 : A \to 0$

▷ Prove subresult $\mathsf{z\_forms\_map} : \{\forall i, j \mid 0\}\ (i = j) \to (\mathsf{z\_fun}\ i) = (\mathsf{z\_fun}\ j)$
$= [\lambda i, j \mid 0]\ [\lambda\_ : i = j]\ \mathsf{ex\_O}\ ((\mathsf{z\_fun}\ i) = (\mathsf{z\_fun}\ j))\ i$

▷ Prove subresult $\mathsf{z'\_forms\_map} : \{\forall x, y \mid A\}\ (x = y) \to (\mathsf{z'\_fun}\ x) = (\mathsf{z'\_fun}\ y)$
$= [\lambda x, y \mid A]\ [\lambda\_ : x = y]\ \mathsf{ex\_A}\ ((\mathsf{z'\_fun}\ x) = (\mathsf{z'\_fun}\ y))\ x$

▷ Let $\mathsf{z\_map} = (\mathsf{z\_fun}, \mathsf{z\_forms\_map} : \mathsf{map}\ 0\ A) : \mathsf{map}\ 0\ A$

▷ Let $\mathsf{z'\_map} = (\mathsf{z'\_fun}, \mathsf{z'\_forms\_map} : \mathsf{map}\ A\ 0) : \mathsf{map}\ A\ 0$

▷ Prove subresult $\mathsf{z\_P}_1 : \{\forall i : 0\}\ (\mathsf{z'\_map} \circ \mathsf{z\_map}\ i) = i$
$= [\lambda i : 0]\ \mathsf{ex\_O}\ ((\mathsf{z'\_map} \circ \mathsf{z\_map}\ i) = i)\ i$

▷ Prove subresult $\mathsf{z\_P}_2 : \{\forall x : A\}\ (\mathsf{z\_map} \circ \mathsf{z'\_map}\ x) = x$
$= [\lambda x : A]\ \mathsf{ex\_A}\ ((\mathsf{z\_map} \circ \mathsf{z'\_map}\ x) = x)\ x$

▷ Prove subresult $\mathsf{z\_forms\_iso} : \mathsf{z\_map} \in (\mathsf{iso}\ 0\ A)$
$= (\mathsf{z'\_map}, \mathsf{pair}\ \mathsf{z\_P}_1\ \mathsf{z\_P}_2 : \mathsf{z\_map} \in (\mathsf{iso}\ 0\ A))$

▷ Prove subresult $\mathsf{z\_iso} : 0.\mathsf{iso}\ A$
$= (\mathsf{z\_map}, \mathsf{z\_forms\_iso} : 0.\mathsf{iso}\ A)$

▷ Prove subresult $\mathsf{Pz} : 0 \leq 0$
$= \mathsf{inr\_is\_true}\ (0 < 0)\ \tau$

▷ Prove subresult $\mathsf{Cz}_2 : (A.\mathsf{is\_decideable\_in}\ B) \to \mathsf{p}_2\ 0\ A$
$= [\lambda\_ : A.\mathsf{is\_decideable\_in}\ B]\ (0, \mathsf{pair\ z\_iso\ Pz} : \mathsf{p}_2\ 0\ A)$

▷ Prove subresult $\mathsf{Cz}_3 : (0.\mathsf{iso}\ A) \to (A \subseteq B) \wedge (\{\forall x : B\}\ x \in A)$
$= [\lambda\_ : 0.\mathsf{iso}\ A]\ \mathsf{pair}\ H\ ([\lambda x : B]\ \mathsf{ex\_O}\ (x \in A)\ (b^{-1}\ x))$

▷ Prove subresult $\mathsf{Cz}_{23} : ((A.\mathsf{is\_decideable\_in}\ B) \to \mathsf{p}_2\ 0\ A) \wedge$
$((0.\mathsf{iso}\ A) \to (A \subseteq B) \wedge (\{\forall x : B\}\ x \in A))$
$= \mathsf{pair}\ \mathsf{Cz}_2\ \mathsf{Cz}_3$

▷ Discharge b

▷ Prove subresult $\mathsf{Cz} : ((0.\mathsf{iso}\ A) \to B.\mathsf{is\_discrete} \to A.\mathsf{is\_decideable\_in}\ B) \wedge$
$((0.\mathsf{iso}\ B) \to ((A.\mathsf{is\_decideable\_in}\ B) \to \mathsf{p}_2\ 0\ A) \wedge$
$((0.\mathsf{iso}\ A) \to (A \subseteq B) \wedge (\{\forall x : B\}\ x \in A)))$
$= \mathsf{pair}\ \mathsf{Cz}_1\ \mathsf{Cz}_{23}$

▷ Discharge H, B, A

▷ Introduce n and suppose $ih : \mathsf{p}_1\ n$

▷ Introduce A, B and suppose $H : A \subseteq B$

▷ Introduce $a : n{+}1.\mathsf{iso}\ A$; suppose $H_1 : B.\mathsf{is\_discrete}$ and introduce $x : B$

▷ Prove subresult $\mathsf{Ps}_{1\text{-}1} : (A \setminus (a\ (\mathsf{last\_}\ n))) \subseteq B$
$= (\mathsf{EXCL}_0\ A\ (a\ (\mathsf{last\_}\ n))).\mathsf{subs\_tran}\ H$

▷ Prove subresult $\mathsf{Ps}_{1\text{-}2} : n.\mathsf{iso}\ (A \setminus (a\ (\mathsf{last\_}\ n)))$
$= \mathsf{EXCL}_4\ a\ (a\ (\mathsf{last\_}\ n))$

▷ Prove subresult $\mathsf{IH}_1 : (A \setminus (a\ (\mathsf{last\_}\ n))).\mathsf{is\_decideable\_in}\ B$
$= (ih\ \mathsf{Ps}_{1\text{-}1}).\mathsf{fst}\ \mathsf{Ps}_{1\text{-}2}\ H_1$

▷ Suppose $H_{1\text{-}1} : x \in (A \setminus (a\ (\mathsf{last\_}\ n)))$

▷ Prove subresult $\mathsf{C}_{1\text{-}1} : (x \in A) \vee (x \notin A)$
$= \mathsf{inl\_is\_true}\ H_{1\text{-}1}.\mathsf{fst}\ (x \notin A)$

▷ Suppose $H_{1\text{-}2} : x \notin (A \setminus (a\ (\mathsf{last\_}\ n)))$

▷ Suppose $H_{1\text{-}2\text{-}1} : (H\ (a\ (\mathsf{last\_}\ n))) = x$

▷ Prove subresult $\mathsf{C}_{1\text{-}2\text{-}1} : (x \in A) \vee (x \notin A)$
$= \mathsf{inl\_is\_true}\ (\mathsf{eq\_closed}|S|A\ H_{1\text{-}2\text{-}1}\ (a\ (\mathsf{last\_}\ n)).\mathsf{ev})\ (x \notin A)$

▷ Suppose $H_{1\text{-}2\text{-}2} : (H\ (a\ (\mathsf{last\_}\ n))) \neq x$

▷ Prove subresult $\mathsf{C}_{1\text{-}2\text{-}2} : (x \in A) \vee ((x \in A) \to \mathsf{false})$
$= \mathsf{inr\_is\_true}\ (x \in A)\ ([\lambda X : x \in A]\ H_{1\text{-}2}\ (\mathsf{pair}\ X\ H_{1\text{-}2\text{-}2}))$

▷ Discharge $H_{1\text{-}2\text{-}2}, H_{1\text{-}2\text{-}1}$

▷ Prove subresult $\mathsf{C_{1\_2}} : (x \in A) \vee (x \notin A)$
$= \mathsf{case}\ (H_1\ (H\ (a\ (\mathsf{last\_}\ n)))\ x)\ \mathsf{C_{1\_2\_1}}\ \mathsf{C_{1\_2\_2}}$

▷ Discharge H_{1_2}, H_{1_1}

▷ Prove subresult $\mathsf{Cs_1} : (x \in A) \vee (x \notin A)$
$= \mathsf{case}\ (\mathsf{IH_1}\ x)\ \mathsf{C_{1\_1}}\ \mathsf{C_{1\_2}}$

▷ Discharge x, H_1, a

▷ Introduce $b : n{+}1.\mathsf{iso}\ B$

▷ Let $\mathsf{b\_n} = b\ (\mathsf{last\_}\ n) : B$

▷ Prove subresult $\mathsf{DB} : B.\mathsf{is\_discrete}$
$= \mathsf{DISC_2}\ (n{+}1, b : B.\mathsf{is\_finite})$

▷ Prove subresult $\mathsf{DA} : A.\mathsf{is\_discrete}$
$= [\lambda x, y : A]\ \mathsf{DB}\ (H\ x)\ (H\ y)$

▷ Let $\mathsf{B'} = B \setminus \mathsf{b\_n} : \mathsf{subset}\ S$

▷ Prove subresult $\mathsf{b_1} : \mathsf{iso}\ ((\mathsf{canon\_subset}\ n{+}1) \setminus (\mathsf{last\_}\ n))\ (B \setminus (b\ (\mathsf{last\_}\ n)))$
$= \mathsf{squeeze\_iso}\ b\ (\mathsf{last\_}\ n)$

▷ Prove subresult $\mathsf{Qn} : (n{+}1 \setminus (\mathsf{last\_}\ n)) = n$
$= \mathsf{CANON_1}\ n$

▷ Prove subresult $\mathsf{b'} : n.\mathsf{iso}\ \mathsf{B'}$
$= \mathsf{ISO_1}\ \mathsf{Qn}\ \mathsf{B'}\ \mathsf{b_1}$

▷ Introduce $a : n{+}1.\mathsf{iso}\ A$

▷ Suppose $H_1 : A.\mathsf{is\_decideable\_in}\ B$

▷ Suppose $H_{1\_1} : \mathsf{b\_n} \in A$

▷ Let $\mathsf{A'} = A \setminus H_{1\_1} : \mathsf{subset}\ S$

▷ Prove subresult $\mathsf{P_{1\_1}} : \mathsf{A'} \subseteq \mathsf{B'}$
$= \mathsf{EXCL_1s}\ H\ H_{1\_1}$

▷ Prove subresult $\mathsf{IH_{23\_1}}$
$: ((\mathsf{A'.is\_decideable\_in\ B'}) \to \mathsf{p_2}\ n\ \mathsf{A'}) \wedge ((n.\mathsf{iso\ A'}) \to \mathsf{A'} = \mathsf{B'})$
$= (ih\ \mathsf{P_{1\_1}}).\mathsf{snd\ b'}$

▷ Prove subresult $\mathsf{P_{1\_2}} : \mathsf{A'.is\_decideable\_in\ B'}$
$= [\lambda x : \mathsf{B'}]\ \mathsf{DEC_3}\ (H_1\ x.\mathsf{ev.fst})\ (\mathsf{DEC_5}\ (\mathsf{DB\ b\_n}\ x.\mathsf{ev.fst}))$

▷ Prove subresult $\mathsf{IH_{2\_1}} : \mathsf{p_2}\ n\ \mathsf{A'}$
$= \mathsf{IH_{23\_1}.fst\ P_{1\_2}}$

▷ Let $\mathsf{m}_1 = \mathsf{IH}_{2\_1\cdot\mathbf{1}} : \mathbb{N}$

▷ Prove subresult $\mathsf{a}_1\text{'} : \mathsf{m}_1.\mathsf{iso}\ \mathsf{A'}$
$= \mathsf{IH}_{2\_1\cdot\mathbf{2}}.\mathsf{fst}$

▷ Prove subresult $\mathsf{Ps}_2 : \mathsf{m}_1{+}1 \leq n{+}1$
$= \mathsf{IH}_{2\_1\cdot\mathbf{2}}.\mathsf{snd}$

▷ Prove subresult $\mathsf{Qm} : \mathsf{m}_1 = (\mathsf{m}_1{+}1 \setminus (\mathsf{last\_}\ \mathsf{m}_1))$
$= (\mathsf{CANON}_1\ \mathsf{m}_1).\mathsf{equal\_subs\_symm}$

▷ Prove subresult $\mathsf{a}_1\text{''} : \mathsf{iso}\ (\mathsf{m}_1{+}1 \setminus (\mathsf{last\_}\ \mathsf{m}_1))\ \mathsf{A'}$
$= \mathsf{ISO}_1\ \mathsf{Qm}\ \mathsf{A'}\ \mathsf{a}_1\text{'}$

▷ Prove subresult $\mathsf{a}_1 : \mathsf{m}_1{+}1.\mathsf{iso}\ A$
$= \mathsf{stretch\_iso}\ (\mathsf{last\_}\ \mathsf{m}_1)\ H_{1\_1}\ (\mathsf{n\_discrete}\ \mathsf{m}_1{+}1)\ \mathsf{DA}\ \mathsf{a}_1\text{''}$

▷ Prove subresult $\mathsf{Cs}_{2\_1} : \mathsf{p}_2\ n{+}1\ A$
$= (\mathsf{m}_1{+}1, \mathsf{pair}\ \mathsf{a}_1\ \mathsf{Ps}_2 : \mathsf{p}_2\ n{+}1\ A)$

▷ Prove subresult $\mathsf{a}\text{'}_1 : n.\mathsf{iso}\ \mathsf{A'}$
$= \mathsf{EXCL}_4\ a\ H_{1\_1}$

▷ Prove subresult $\mathsf{IH}_{3\_1} : \mathsf{A'} = \mathsf{B'}$
$= \mathsf{IH}_{23\_1}.\mathsf{snd}\ \mathsf{a}\text{'}_1$

▷ Introduce $x : B$

▷ Prove subresult $\mathsf{Cs}_{2\_2\_1} : (\mathsf{b\_n} = x) \to x \in A$
$= [\lambda X : \mathsf{b\_n} = x]\ \mathsf{eq\_closed}|S|A\ X\ H_{1\_1}$

▷ Suppose $X : \mathsf{b\_n} \neq x$

▷ Prove subresult $\mathsf{Ps}_{1\_3\_1} : x \in \mathsf{B'}$
$= \mathsf{pair}\ x.\mathsf{ev}\ X$

▷ Prove subresult $\mathsf{Ps}_{1\_3\_2} : \mathsf{A'} \subseteq A$
$= \mathsf{EXCL}_0\ A\ H_{1\_1}$

▷ Prove subresult $\mathsf{Ps}_{1\_3\_3} : \mathsf{B'} \subseteq A$
$= \mathsf{IH}_{3\_1}.\mathsf{snd}.\mathsf{subs\_tran}\ \mathsf{Ps}_{1\_3\_2}$

▷ Prove subresult $\mathsf{Cs}_{2\_2\_2} : x \in A$
$= \mathsf{Ps}_{1\_3\_3}\ (\mathsf{make}\ \mathsf{Ps}_{1\_3\_1})$

▷ Discharge X

▷ Prove subresult $\mathsf{Ps}_{1\_3} : x \in A$
$= \mathsf{case}\ (\mathsf{DB}\ \mathsf{b\_n}\ x)\ \mathsf{Cs}_{2\_2\_1}\ \mathsf{Cs}_{2\_2\_2}$

▷ Discharge x

▷ Prove subresult $\mathsf{Cs}_{3\text{-}1} : A = B$
$= \mathsf{pair}\ H\ \mathsf{Ps}_{1\text{-}3}$

▷ Discharge $H_{1\text{-}1}$

▷ Suppose $H_2 : \mathsf{b\_n} \notin A$

▷ Prove subresult $\mathsf{P}_{2\text{-}1} : A \subseteq \mathsf{B'}$
$= [\lambda x : A]\ \mathsf{pair}\ (H\ x)\ ([\lambda X : \mathsf{b\_n}.(\mathsf{equal\_in}\ S)\ x]\ H_2\ (\mathsf{eq\_closed}\ X.\mathsf{symm}\ x.\mathsf{ev}))$

▷ Prove subresult $\mathsf{IH}_{23\text{-}2}$
$: ((A.\mathsf{is\_decideable\_in}\ \mathsf{B'}) \to \mathsf{p}_2\ n\ A) \wedge ((n.\mathsf{iso}\ A) \to A = \mathsf{B'})$
$= (ih\ \mathsf{P}_{2\text{-}1}).\mathsf{snd}\ \mathsf{b'}$

▷ Prove subresult $\mathsf{P}_{2\text{-}2} : A.\mathsf{is\_decideable\_in}\ \mathsf{B'}$
$= [\lambda x : \mathsf{B'}]\ H_1\ x.\mathsf{ev}.\mathsf{fst}$

▷ Prove subresult $\mathsf{IH}_{2\text{-}2} : \mathsf{p}_2\ n\ A$
$= \mathsf{IH}_{23\text{-}2}.\mathsf{fst}\ \mathsf{P}_{2\text{-}2}$

▷ Let $\mathsf{m}_2 = \mathsf{IH}_{2\text{-}2}.\mathbf{1} : \mathbb{N}$

▷ Prove subresult $\mathsf{a}_2 : \mathsf{m}_2.\mathsf{iso}\ A$
$= \mathsf{IH}_{2\text{-}2}.\mathbf{2}.\mathsf{fst}$

▷ Prove subresult $\mathsf{Ps}_{2\text{-}1} : \mathsf{m}_2 \leq n$
$= \mathsf{IH}_{2\text{-}2}.\mathbf{2}.\mathsf{snd}$

▷ Prove subresult $\mathsf{Ps}_{2\text{-}2} : n \leq n{+}1$
$= \mathsf{inl\_is\_true}\ (\mathsf{LESS}_1\ n)\ (n = n{+}1)$

▷ Prove subresult $\mathsf{Ps}_{2\text{-}3} : \mathsf{m}_2 \leq n{+}1$
$= \mathsf{Ps}_{2\text{-}1}.\mathsf{leq\_than\_tran}\ \mathsf{Ps}_{2\text{-}2}$

▷ Prove subresult $\mathsf{Cs}_{2\text{-}2} : \mathsf{p}_2\ n{+}1\ A$
$= (\mathsf{m}_2, \mathsf{pair}\ \mathsf{a}_2\ \mathsf{Ps}_{2\text{-}3} : \mathsf{p}_2\ n{+}1\ A)$

▷ Prove subresult $\mathsf{Ps}_{2\text{-}4} : \mathsf{iso}\ \mathsf{m}_2\ n{+}1$
$= \mathsf{a}_2.\mathsf{iso\_tran}\ a.\mathsf{iso\_symm}$

▷ Prove subresult $\mathsf{Ps}_{2\text{-}5} : n{+}1 = \mathsf{m}_2$
$= \mathsf{EQSIZE}_1\ \mathsf{Ps}_{2\text{-}4}$

▷ Prove subresult $\mathsf{Ps}_{2\text{-}6} : \mathsf{m}_2 < n{+}1$
$= \mathsf{LEQ}_1\ \mathsf{m}_2\ n\ \mathsf{Ps}_{2\text{-}1}$

▷ Prove subresult $\mathsf{Ps}_{2\text{-}7} : \mathsf{m}_2 < \mathsf{m}_2$
$= \mathsf{nat\_eq\_resp}\ \mathsf{Ps}_{2\text{-}5}\ \mathsf{m}_2 < \mathsf{Ps}_{2\text{-}6}$

▷ Prove subresult $\mathsf{Cs}_{3\text{-}2} : A = B$
$= \mathsf{ex\_falso}\ (A = B)\ (\mathsf{less\_than\_arefl}\ \mathsf{m}_2\ \mathsf{Ps}_{2\text{-}7})$

▷ Discharge H_2

▷ Prove subresult $\mathsf{Cs_2}$ **:** $\mathsf{p_2}\ n{+}1\ A$

$= \mathsf{case}\ (H_1\ \mathsf{b\_n})\ \mathsf{Cs_{2\_1}}\ \mathsf{Cs_{2\_2}}$

▷ Prove subresult $\mathsf{Cs_3'}$ **:** $A = B$

$= \mathsf{case}\ (\mathsf{Cs_1}\ a\ \mathsf{DB}\ \mathsf{b\_n})\ \mathsf{Cs_{3\_1}}\ \mathsf{Cs_{3\_2}}$

▷ Discharge H_1

▷ Prove subresult $\mathsf{Cs_3}$ **:** $A = B$

$= \mathsf{Cs_3'}\ (\mathsf{Cs_1}\ a\ \mathsf{DB})$

▷ Discharge a

▷ Prove subresult $\mathsf{Cs_{23}}$
: $((A.\mathsf{is\_decideable\_in}\ B) \to \mathsf{p_2}\ n{+}1\ A) \wedge ((n{+}1.\mathsf{iso}\ A) \to A = B)$

$= \mathsf{pair}\ \mathsf{Cs_2}\ \mathsf{Cs_3}$

▷ Discharge b

▷ Prove subresult Cs
: $((n{+}1.\mathsf{iso}\ A) \to B.\mathsf{is\_discrete} \to \{\forall x : B\}\ (x \in A) \vee (x \notin A)) \wedge$
$((n{+}1.\mathsf{iso}\ B) \to$
$((A.\mathsf{is\_decideable\_in}\ B) \to \mathsf{p_2}\ n{+}1\ A) \wedge ((n{+}1.\mathsf{iso}\ A) \to A = B))$

$= \mathsf{pair}\ \mathsf{Cs_1}\ \mathsf{Cs_{23}}$

▷ Discharge H, B, A, ih, n

▷ Prove $\mathsf{FIN_0}$ **:** $\{\Pi n\}\ \mathsf{p_1}\ n$

$= \mathsf{N\_elim}\ \mathsf{p_1}\ \mathsf{Cz}\ \mathsf{Cs}$

▷ Introduce A, B

▷ Prove $\mathsf{FIN_1}$ **:** $B.\mathsf{is\_finite} \to (A.\mathsf{decideable\_subs}\ B) \to A \preccurlyeq B$

$= [\lambda H_1 : B.\mathsf{is\_finite}]\ [\lambda H_2 : A.\mathsf{decideable\_subs}\ B]$
$[\delta \mathsf{P} = ((\mathsf{FIN_0}\ H_{1\cdot\mathbf{1}}\ H_2.\mathsf{snd}).\mathsf{snd}\ H_{1\cdot\mathbf{2}}).\mathsf{fst}\ H_2.\mathsf{fst}]$
$(\mathsf{P}_{\cdot\mathbf{1}}, H_{1\cdot\mathbf{1}}, \mathsf{pair_3}\ \mathsf{P}_{\cdot\mathbf{2}}.\mathsf{fst}\ H_{1\cdot\mathbf{2}}\ \mathsf{P}_{\cdot\mathbf{2}}.\mathsf{snd} : A \preccurlyeq B)$

▷ Prove $\mathsf{FIN_2}$ **:** $(A = B) \to A \cong B$

$= [\lambda Q : A = B]\ \mathsf{ISO_1}\ Q\ A\ (\mathsf{iso\_refl}\ A)$

▷ Prove $\mathsf{FIN_3}$ **:** $(A \subseteq B) \to A.\mathsf{is\_finite} \to (A \cong B) \to A = B$

$= [\lambda H_1 : A \subseteq B]\ [\lambda H_2 : A.\mathsf{is\_finite}]\ [\lambda H_3 : A \cong B]$
$((\mathsf{FIN_0}\ H_{2\cdot\mathbf{1}}\ H_1).\mathsf{snd}\ (H_{2\cdot\mathbf{2}}.\mathsf{iso\_tran}\ H_3.\mathsf{iso\_symm})).\mathsf{snd}\ H_{2\cdot\mathbf{2}}$

▷ Prove $\mathsf{FIN_4}$ **:** $(A = B) \to A.\mathsf{is\_finite} \to B.\mathsf{is\_finite}$

$= [\lambda Q : A = B]\ [\lambda H : A.\mathsf{is\_finite}]\ \mathsf{EQSIZE_0}\ (\mathsf{FIN_2}\ Q)\ H$

▷ Discharge

$B, A, \mathsf{Cs}, \mathsf{Cs}_{23}, \mathsf{Cs}_3, \mathsf{Cs}_3\text{'}, \mathsf{Cs}_2, \mathsf{Cs}_{3\text{–}2}, \mathsf{Ps}_{2\text{–}7}, \mathsf{Ps}_{2\text{–}6}, \mathsf{Ps}_{2\text{–}5}, \mathsf{Ps}_{2\text{–}4}, \mathsf{Cs}_{2\text{–}2}, \mathsf{Ps}_{2\text{–}3},$
$\mathsf{Ps}_{2\text{–}2}, \mathsf{Ps}_{2\text{–}1}, \mathsf{a}_2, \mathsf{m}_2, \mathsf{IH}_{2\text{–}2}, \mathsf{P}_{2\text{–}2}, \mathsf{IH}_{23\text{–}2}, \mathsf{P}_{2\text{–}1}, \mathsf{Cs}_{3\text{–}1}, \mathsf{Ps}_{1\text{–}3}, \mathsf{Cs}_{2\text{–}2\text{–}2}, \mathsf{Ps}_{1\text{–}3\text{–}3},$
$\mathsf{Ps}_{1\text{–}3\text{–}2}, \mathsf{Ps}_{1\text{–}3\text{–}1}, \mathsf{Cs}_{2\text{–}2\text{–}1}, \mathsf{IH}_{3\text{–}1}, \mathsf{a}\text{'}_1, \mathsf{Cs}_{2\text{–}1}, \mathsf{a}_1, \mathsf{a}_1\text{''}, \mathsf{Qm}, \mathsf{Ps}_2, \mathsf{a}_1\text{'}, \mathsf{m}_1, \mathsf{IH}_{2\text{–}1}, \mathsf{P}_{1\text{–}2},$
$\mathsf{IH}_{23\text{–}1}, \mathsf{P}_{1\text{–}1}, \mathsf{A}\text{'}, \mathsf{b}\text{'}, \mathsf{Qn}, \mathsf{b}_1, \mathsf{B}\text{'}, \mathsf{DA}, \mathsf{DB}, \mathsf{b\_n}, \mathsf{Cs}_1, \mathsf{C}_{1\text{–}2}, \mathsf{C}_{1\text{–}2\text{–}2}, \mathsf{C}_{1\text{–}2\text{–}1}, \mathsf{C}_{1\text{–}1}, \mathsf{IH}_1,$
$\mathsf{Ps}_{1\text{–}2}, \mathsf{Ps}_{1\text{–}1}, \mathsf{Cz}, \mathsf{Cz}_{23}, \mathsf{Cz}_3, \mathsf{Cz}_2, \mathsf{Pz}, \mathsf{z\_iso}, \mathsf{z\_forms\_iso}, \mathsf{z\_P}_2, \mathsf{z\_P}_1, \mathsf{z}\text{'}\mathsf{\_map},$
$\mathsf{z\_map}, \mathsf{z}\text{'}\mathsf{\_forms\_map}, \mathsf{z\_forms\_map}, \mathsf{z}\text{'}\mathsf{\_fun}, \mathsf{z\_fun}, \mathsf{ex\_A}, \mathsf{Cz}_1, \mathsf{p}_1, \mathsf{p}_2, S$

B.4.3.5 Results relating decideability and finite quantification

▷ Construct by refinement FIN_5
: $\{\forall n \,|\, \mathbb{N}\}\ \{\forall P : \mathsf{subset}\ n\}\ (\{\forall i\}\ (P.\mathsf{pred}\ i).\mathsf{or\_not}) \to (\langle \exists i \rangle\ P.\mathsf{pred}\ i).\mathsf{or\_not}$
($+\underline{1}$, $+1$, pred, subset, $\underline{0}$, DEC_4, symm, eq_closed, inr, equal_in, inl, CANON_2, $\vee$, case, pair, DEC_2, resp, $\mathbb{N}$, $=$, or_not, LESS_0, 0, or_not_is_false, N_elim)

▷ Construct by refinement FIN_6
: $\{\forall n \,|\, \mathbb{N}\}\ \{\forall P : \mathsf{subset}\ n\}\ (\{\forall i\}\ (P.\mathsf{pred}\ i).\mathsf{or\_not}) \to (\{\forall i\}\ P.\mathsf{pred}\ i).\mathsf{or\_not}$
($+\underline{1}$, $+1$, pred, subset, $\underline{0}$, DEC_3, pair, snd, eq_closed, equal_in, fst, CANON_2, case, $\wedge$, DEC_2, resp, $\mathbb{N}$, $=$, or_not, LESS_0, 0, ex_falso, or_not_is_true, N_elim)

▷ Introduce S and A

▷ Introduce A_Fin : $A.\mathsf{is\_finite}$

▷ Let $\mathsf{n} = A\_Fin._{\mathbf{1}}$: $\mathbb{N}$

▷ Let $\psi = A\_Fin._{\mathbf{2}}$: $\mathsf{iso}\ \mathsf{n}\ A$

▷ Introduce P : $\mathsf{subset}\ A$

▷ Construct by refinement FIN_7 : $(\langle \exists x : A \rangle\ x \in P) \leftrightarrow (\langle \exists i : \mathsf{n} \rangle\ (\psi\ i) \in P)$
(ψ, $\in$, n, iso_compose_inverse, identity, is_map, $^{-1}$, $\circ$, symm, subset_forms_set, $=$, set, canon_subset, $\mathbb{N}$, eq_closed, pair)

▷ Discharge $P, \psi, \mathsf{n}, A\_Fin, A, S$

B.5 Algebraic structures

B.5.1 Groups and fields

B.5.1.1 Groups

▷ Explicitly overload the identifier $\circ$

▷ Explicitly overload the identifier $^{-1}$

▷ Define $\mathsf{group\_axioms} = [\lambda G \,|\, \mathsf{set}]\ [\lambda id : G]\ [\lambda \circ : \mathsf{map}_2\ G\ G\ G]\ [\lambda^{-1} : \mathsf{map}\ G\ G]$

$\bigwedge_3 (\{\forall x, y, z : G\}\ ((x \circ y) \circ z) = (x \circ (y \circ z)))$

$((\{\forall x : G\}\ (x \circ id) = x) \wedge (\{\Pi x : G\}\ (id \circ x) = x))$

$(\{\forall x : G\}\ (x \circ x^{-1}) = id)$

$: \{\Pi G \,|\, \mathsf{set}\}\ G \to (\mathsf{map}_2\ G\ G\ G) \to (\mathsf{map}\ G\ G) \to \mathsf{prop}$

▷ Define $\mathsf{group} = \langle \Sigma G : \mathsf{set} \rangle\ \langle \Sigma id : G \rangle\ \langle \Sigma \circ : \mathsf{map}_2\ G\ G\ G \rangle\ \langle \Sigma^{-1} : \mathsf{map}\ G\ G \rangle$

$\mathsf{group\_axioms}\ id\ \circ\ ^{-1} : \mathsf{Type}_1$

▷ Define coercion $\mathsf{car} = [\lambda G]\ G_{.\mathbf{1}} : \mathsf{group} \to \mathsf{set}$

▷ Unless otherwise specified, by default $G : \mathsf{group}$

▷ Introduce $G \mid \mathsf{group}$

▷ Define $\mathsf{id} = G_{.\mathbf{2}\cdot\mathbf{1}} : G$

▷ Define $\circ = G_{.\mathbf{2}\cdot\mathbf{2}\cdot\mathbf{1}} : \mathsf{map}_2\ G\ G\ G$

▷ Define $^{-1} = G_{.\mathbf{2}\cdot\mathbf{2}\cdot\mathbf{2}\cdot\mathbf{1}} : \mathsf{map}\ G\ G$

▷ Introduce $x_1, x_2 \mid G$; suppose $Qx : x_1 = x_2$; introduce $x, y, z : G$; $y_1, y_2 \mid G$ and suppose $Qy : y_1 = y_2$

▷ Prove $\mathsf{o\_resp} : (x_1 \circ y_1) = (x_2 \circ y_2)$
$= \mathsf{resp}_2 \circ Qx\ Qy$

▷ Prove $\mathsf{o}_1 : (x_1 \circ y) = (x_2 \circ y)$
$= \mathsf{resps}_1 \circ Qx\ y$

▷ Prove $\mathsf{o}_2 : (x \circ y_1) = (x \circ y_2)$
$= \mathsf{resps}_2 \circ x\ Qy$

▷ Prove $\mathsf{inv\_resp} : {x_1}^{-1} = {x_2}^{-1}$
$= \mathsf{resp}\ ^{-1}\ Qx$

▷ Prove $\mathsf{inv}_1 : {x_1}^{-1} = {x_2}^{-1}$
$= \mathsf{inv\_resp}$

▷ Prove $\mathsf{o\_assoc} : ((x \circ y) \circ z) = (x \circ (y \circ z))$
$= G.\mathbf{2}.\mathbf{2}.\mathbf{2}.\mathbf{2}.\mathsf{fst}_3\ x\ y\ z$

▷ Prove $\mathsf{o\_id} : (x \circ \mathsf{id}) = x$
$= G.\mathbf{2}.\mathbf{2}.\mathbf{2}.\mathbf{2}.\mathsf{snd}_3.\mathsf{fst}\ x$

▷ Prove $\mathsf{id\_o} : (\mathsf{id} \circ x) = x$
$= G.\mathbf{2}.\mathbf{2}.\mathbf{2}.\mathbf{2}.\mathsf{snd}_3.\mathsf{snd}\ x$

▷ Prove $\mathsf{o\_inv} : (x \circ x^{-1}) = \mathsf{id}$
$= G.\mathbf{2}.\mathbf{2}.\mathbf{2}.\mathbf{2}.\mathsf{thd}_3\ x$

▷ Discharge $Qy, y_2, y_1, z, y, x, Qx, x_2, x_1, G$

B.5.1.2 Results concerning groups

▷ Introduce $G \mid \mathsf{group}$ and $x, y : G$

▷ Prove subresult $\mathsf{P}_1 : x^{-1-1} = (\mathsf{id} \circ x^{-1-1})$
$= (\mathsf{id\_o}\ x^{-1-1}).\mathsf{symm}$

▷ Prove subresult $\mathsf{P}_2 : (\mathsf{id} \circ x^{-1-1}) = ((x \circ x^{-1}) \circ x^{-1-1})$
$= ((\mathsf{o\_inv}\ x).\mathsf{o}_1\ x^{-1-1}).\mathsf{symm}$

▷ Prove subresult $\mathsf{P}_3 : ((x \circ x^{-1}) \circ x^{-1-1}) = (x \circ (x^{-1} \circ x^{-1-1}))$
$= x.\mathsf{o\_assoc}\ x^{-1}\ x^{-1-1}$

▷ Prove subresult $\mathsf{P}_4 : (x \circ (x^{-1} \circ x^{-1-1})) = (x \circ \mathsf{id})$
$= x.\mathsf{o}_2\ x^{-1}.\mathsf{o\_inv}$

▷ Prove subresult $\mathsf{P}_5 : (x \circ \mathsf{id}) = x$
$= x.\mathsf{o\_id}$

▷ Prove $\mathsf{inv\_inv} : x^{-1-1} = x$
$= \mathsf{P}_1.\mathsf{tran}\ (\mathsf{P}_2.\mathsf{tran}\ (\mathsf{P}_3.\mathsf{tran}\ (\mathsf{P}_4.\mathsf{tran}\ \mathsf{P}_5)))$

▷ Discharge $\mathsf{P}_5, \mathsf{P}_4, \mathsf{P}_3, \mathsf{P}_2, \mathsf{P}_1$

▷ Prove subresult $\mathsf{P}_1 : (x^{-1} \circ x) = (x^{-1} \circ x^{-1-1})$
$= x^{-1}.\mathsf{o}_2\ \mathsf{inv\_inv.symm}$

▷ Prove subresult $\mathsf{P}_2 : (x^{-1} \circ x^{-1-1}) = \mathsf{id}$
$= x^{-1}.\mathsf{o\_inv}$

▷ Prove $\mathsf{inv\_o} : (x^{-1} \circ x) = \mathsf{id}$
$= \mathsf{P}_1.\mathsf{tran}\ \mathsf{P}_2$

▷ Discharge $\mathsf{P}_2, \mathsf{P}_1, y, x$

▷ Introduce $x, y : G$

▷ Suppose $Q : x = y$

▷ Prove subresult $\mathsf{Q}_{1\_1} : (x \circ y^{-1}) = (x \circ x^{-1})$
$= x.\mathsf{o}_2\ (\mathsf{inv\_resp}\ Q.\mathsf{symm})$

▷ Prove subresult $\mathsf{Q}_{1\_2} : (x \circ x^{-1}) = \mathsf{id}$
$= x.\mathsf{o\_inv}$

▷ Prove subresult $\mathsf{P}_1 : (x \circ y^{-1}) = \mathsf{id}$
$= \mathsf{Q}_{1\_1}.\mathsf{tran}\ \mathsf{Q}_{1\_2}$

▷ Prove subresult $\mathsf{Q}_{2\_1} : (x^{-1} \circ y) = (y^{-1} \circ y)$
$= (\mathsf{inv\_resp}\ Q).\mathsf{o}_1\ y$

▷ Prove subresult $\mathsf{Q}_{2\_2} : (y^{-1} \circ y) = \mathsf{id}$
$= \mathsf{inv\_o}\ y$

▷ Prove subresult $\mathsf{P}_2 : (x^{-1} \circ y) = \mathsf{id}$
$= \mathsf{Q}_{2\_1}.\mathsf{tran}\ \mathsf{Q}_{2\_2}$

▷ Discharge Q

▷ Suppose $Q : (x \circ y^{-1}) = \mathsf{id}$

▷ Prove subresult $\mathsf{Q}_{3\_1} : x = (x \circ \mathsf{id})$
$= x.\mathsf{o\_id.symm}$

▷ Prove subresult $\mathsf{Q}_{3\_2} : (x \circ \mathsf{id}) = (x \circ (y^{-1} \circ y))$
$= (x.\mathsf{o}_2\ (\mathsf{inv\_o}\ y)).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_{3\_3} : (x \circ (y^{-1} \circ y)) = ((x \circ y^{-1}) \circ y)$
$= (\mathsf{o\_assoc}\ x\ y^{-1}\ y).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_{3\_4} : ((x \circ y^{-1}) \circ y) = (\mathsf{id} \circ y)$
$= Q.\mathsf{o}_1\ y$

▷ Prove subresult $\mathsf{Q}_{3\_5} : (\mathsf{id} \circ y) = y$
$= \mathsf{id\_o}\ y$

▷ Prove subresult $\mathsf{P}_3 : x = y$
$= (\mathsf{Q}_{3\_1}.\mathsf{tran}\ \mathsf{Q}_{3\_2}).\mathsf{tran}\ (\mathsf{Q}_{3\_3}.\mathsf{tran}\ (\mathsf{Q}_{3\_4}.\mathsf{tran}\ \mathsf{Q}_{3\_5}))$

▷ Discharge Q

▷ Suppose $Q : (x^{-1} \circ y) = \mathsf{id}$

▷ Prove subresult $\mathsf{Q}_{4\_1} : y = (\mathsf{id} \circ y)$
$= (\mathsf{id\_o}\ y).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_{4\text{–}2} : (\mathsf{id} \circ y) = ((x \circ x^{-1}) \circ y)$
$= (x.\mathsf{o\_inv.o}_1\ y).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_{4\text{–}3} : ((x \circ x^{-1}) \circ y) = (x \circ (x^{-1} \circ y))$
$= \mathsf{o\_assoc}\ x\ x^{-1}\ y$

▷ Prove subresult $\mathsf{Q}_{4\text{–}4} : (x \circ (x^{-1} \circ y)) = (x \circ \mathsf{id})$
$= x.\mathsf{o}_2\ Q$

▷ Prove subresult $\mathsf{Q}_{4\text{–}5} : (x \circ \mathsf{id}) = x$
$= x.\mathsf{o\_id}$

▷ Prove subresult $\mathsf{P}_4 : x = y$
$= ((\mathsf{Q}_{4\text{–}1}.\mathsf{tran}\ \mathsf{Q}_{4\text{–}2}).\mathsf{tran}\ (\mathsf{Q}_{4\text{–}3}.\mathsf{tran}\ (\mathsf{Q}_{4\text{–}4}.\mathsf{tran}\ \mathsf{Q}_{4\text{–}5}))).\mathsf{symm}$

▷ Discharge Q

▷ Prove $\mathsf{GROUP}_1 : (x = y) \leftrightarrow ((x \circ y^{-1}) = \mathsf{id})$
$= \mathsf{pair}\ \mathsf{P}_1\ \mathsf{P}_3$

▷ Prove $\mathsf{GROUP}_2 : (x = y) \leftrightarrow ((x^{-1} \circ y) = \mathsf{id})$
$= \mathsf{pair}\ \mathsf{P}_2\ \mathsf{P}_4$

▷ Discharge $\mathsf{P}_4, \mathsf{Q}_{4\text{–}5}, \mathsf{Q}_{4\text{–}4}, \mathsf{Q}_{4\text{–}3}, \mathsf{Q}_{4\text{–}2}, \mathsf{Q}_{4\text{–}1}, \mathsf{P}_3, \mathsf{Q}_{3\text{–}5}, \mathsf{Q}_{3\text{–}4}, \mathsf{Q}_{3\text{–}3}, \mathsf{Q}_{3\text{–}2}, \mathsf{Q}_{3\text{–}1}, \mathsf{P}_2,$
$\mathsf{Q}_{2\text{–}2}, \mathsf{Q}_{2\text{–}1}, \mathsf{P}_1, \mathsf{Q}_{1\text{–}2}, \mathsf{Q}_{1\text{–}1}, y, x$

▷ Introduce $x, y : G$

▷ Prove subresult $\mathsf{P}_1 : ((x^{-1-1} \circ y) = \mathsf{id}) \to (x \circ y) = \mathsf{id}$
$= [\lambda Q : (x^{-1-1} \circ y) = \mathsf{id}]\ (x.\mathsf{inv\_inv.symm.o}_1\ y).\mathsf{tran}\ Q$

▷ Prove subresult $\mathsf{P}_2 : ((x \circ y) = \mathsf{id}) \to (x^{-1-1} \circ y) = \mathsf{id}$
$= [\lambda Q : (x \circ y) = \mathsf{id}]\ (x.\mathsf{inv\_inv.o}_1\ y).\mathsf{tran}\ Q$

▷ Prove subresult $\mathsf{Q}_1 : ((x^{-1-1} \circ y) = \mathsf{id}) \leftrightarrow ((x \circ y) = \mathsf{id})$
$= \mathsf{pair}\ \mathsf{P}_1\ \mathsf{P}_2$

▷ Prove $\mathsf{GROUP}_3 : (x^{-1} = y) \leftrightarrow ((x \circ y) = \mathsf{id})$
$= (\mathsf{GROUP}_2\ x^{-1}\ y).\mathsf{iff\_tran}\ \mathsf{Q}_1$

▷ Discharge $\mathsf{Q}_1, \mathsf{P}_2, \mathsf{P}_1, y, x$

▷ Introduce $x, y : G$

▷ Prove subresult P_1
$: (((x \circ y) \circ (y^{-1} \circ x^{-1})) = \mathsf{id}) \to (x \circ y)^{-1} = (y^{-1} \circ x^{-1})$
$= (\mathsf{GROUP}_3\ (x \circ y)\ (y^{-1} \circ x^{-1})).\mathsf{snd}$

▷ Prove subresult $\mathsf{Q}_1 : ((x \circ y) \circ (y^{-1} \circ x^{-1})) = (x \circ (y \circ (y^{-1} \circ x^{-1})))$
$= \mathsf{o\_assoc}\ x\ y\ (y^{-1} \circ x^{-1})$

▷ Prove subresult $\mathsf{Q}_{2\text{-}1} : (y \circ (y^{-1} \circ x^{-1})) = ((y \circ y^{-1}) \circ x^{-1})$
$= (\mathsf{o\_assoc}\ y\ y^{-1}\ x^{-1}).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_{2\text{-}2} : ((y \circ y^{-1}) \circ x^{-1}) = (\mathsf{id} \circ x^{-1})$
$= y.\mathsf{o\_inv}.\mathsf{o}_1\ x^{-1}$

▷ Prove subresult $\mathsf{Q}_{2\text{-}3} : (\mathsf{id} \circ x^{-1}) = x^{-1}$
$= \mathsf{id\_o}\ x^{-1}$

▷ Prove subresult $\mathsf{Q}_2 : (x \circ (y \circ (y^{-1} \circ x^{-1}))) = (x \circ x^{-1})$
$= x.\mathsf{o}_2\ (\mathsf{Q}_{2\text{-}1}.\mathsf{tran}\ (\mathsf{Q}_{2\text{-}2}.\mathsf{tran}\ \mathsf{Q}_{2\text{-}3}))$

▷ Prove subresult $\mathsf{Q}_3 : (x \circ x^{-1}) = \mathsf{id}$
$= x.\mathsf{o\_inv}$

▷ Prove $\mathsf{inv\_of\_o} : (x \circ y)^{-1} = (y^{-1} \circ x^{-1})$
$= \mathsf{P}_1\ (\mathsf{Q}_1.\mathsf{tran}\ (\mathsf{Q}_2.\mathsf{tran}\ \mathsf{Q}_3))$

▷ Discharge $\mathsf{Q}_3, \mathsf{Q}_2, \mathsf{Q}_{2\text{-}3}, \mathsf{Q}_{2\text{-}2}, \mathsf{Q}_{2\text{-}1}, \mathsf{Q}_1, \mathsf{P}_1, y, x$

▷ Introduce $x \mid G$ and suppose $H : (x \circ x) = x$

▷ Prove subresult $\mathsf{Q}_1 : x = (x \circ \mathsf{id})$
$= x.\mathsf{o\_id}.\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_2 : (x \circ \mathsf{id}) = (x \circ (x \circ x^{-1}))$
$= x.\mathsf{o}_2\ x.\mathsf{o\_inv}.\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_3 : (x \circ (x \circ x^{-1})) = ((x \circ x) \circ x^{-1})$
$= (\mathsf{o\_assoc}\ x\ x\ x^{-1}).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_4 : ((x \circ x) \circ x^{-1}) = (x \circ x^{-1})$
$= H.\mathsf{o}_1\ x^{-1}$

▷ Prove subresult $\mathsf{Q}_5 : (x \circ x^{-1}) = \mathsf{id}$
$= x.\mathsf{o\_inv}$

▷ Prove $\mathsf{GROUP}_4 : x = \mathsf{id}$
$= (\mathsf{Q}_1.\mathsf{tran}\ \mathsf{Q}_2).\mathsf{tran}\ (\mathsf{Q}_3.\mathsf{tran}\ (\mathsf{Q}_4.\mathsf{tran}\ \mathsf{Q}_5))$

▷ Discharge $\mathsf{Q}_5, \mathsf{Q}_4, \mathsf{Q}_3, \mathsf{Q}_2, \mathsf{Q}_1, H, x, G$

B.5.1.3 Abelian groups

▷ Define $\mathsf{abelian\_group\_axioms} = [\lambda G]\ \{\forall x, y : G\}\ (x \circ y) = (y \circ x)$
$: \mathsf{group} \to \mathsf{Type}_0$

▷ Define $\mathsf{abelian\_group} = \langle \Sigma G \rangle\ \mathsf{abelian\_group\_axioms}\ G : \mathsf{Type}_1$

▷ Define coercion $\mathsf{unab\_group} = [\lambda G : \mathsf{abelian\_group}]\ G.\mathbf{1}$

$\quad : \mathsf{abelian\_group} \to \mathsf{group}$

▷ Allow $+$ to be written infix

▷ Allow $-$ to be written prefix

▷ Explicitly overload the identifier $-$

▷ Allow $-$ to be written infix

▷ Introduce $G \mid \mathsf{abelian\_group}$

▷ Define $\mathbf{0} = \mathsf{id}|G : G$

▷ Define $+ = \circ|G : \mathsf{map}_2\ G\ G\ G$

▷ Define $- = {}^{-1}|G : \mathsf{map}\ G\ G$

▷ Prove $\mathsf{minus\_forms\_map}_2 : ([\lambda x, y : G]\ x + (-y)).\mathsf{is\_map}_2$

$\quad = [\lambda x_1, x_2 \mid G]\ [\lambda Qx : x_1 = x_2]\ [\lambda y_1, y_2 \mid G]\ [\lambda Qy : y_1 = y_2]$

$\quad\quad +.\mathsf{resp}_2\ Qx\ (-.\mathsf{resp}\ Qy)$

▷ Define $- = ([\lambda x, y : G]\ x + (-y), \mathsf{minus\_forms\_map}_2 : \mathsf{map}_2\ G\ G\ G)$

$\quad : \mathsf{map}_2\ G\ G\ G$

▷ Introduce $x_1, x_2 \mid G$; suppose $Qx : x_1 = x_2$; introduce $x, y, z : G$; $y_1, y_2 \mid G$ and suppose $Qy : y_1 = y_2$

▷ Prove $\mathsf{plus\_resp} : (x_1 + y_1) = (x_2 + y_2)$

$\quad = \mathsf{o\_resp}\ Qx\ Qy$

▷ Prove $\mathsf{plus}_1 : (x_1 + y) = (x_2 + y)$

$\quad = \mathsf{o}_1\ Qx\ y$

▷ Prove $\mathsf{plus}_2 : (x + y_1) = (x + y_2)$

$\quad = \mathsf{o}_2\ x\ Qy$

▷ Prove $\mathsf{neg\_resp} : (-x_1) = (-x_2)$

$\quad = \mathsf{inv\_resp}\ Qx$

▷ Prove $\mathsf{neg}_1 : (-x_1) = (-x_2)$

$\quad = \mathsf{neg\_resp}$

▷ Prove $\mathsf{minus\_resp} : (x_1 - y_1) = (x_2 - y_2)$

$\quad = \mathsf{resp}_2\ -\ Qx\ Qy$

▷ Prove $\mathsf{minus}_1 : (x_1 - y) = (x_2 - y)$

$\quad = \mathsf{resps}_1\ -\ Qx\ y$

▷ Prove minus_2 : $(x - y_1) = (x - y_2)$
= $\mathsf{resps}_2\ -\ x\ Qy$

▷ Prove plus_assoc : $((x + y) + z) = (x + (y + z))$
= o_assoc $x\ y\ z$

▷ Prove plus_comm : $(x + y) = (y + x)$
= $G.\mathbf{2}\ x\ y$

▷ Prove plus_ze : $(x + \mathbf{0}) = x$
= o_id x

▷ Prove ze_plus : $(\mathbf{0} + x) = x$
= id_o x

▷ Prove plus_neg : $(x + (-x)) = \mathbf{0}$
= o_inv x

▷ Prove minus_self : $(x - x) = \mathbf{0}$
= plus_neg

▷ Discharge $Qy, y_2, y_1, z, y, x, Qx, x_2, x_1, G$

B.5.1.4 Results concerning abelian groups

▷ Introduce G | abelian_group and $w, x, y, z : G$

▷ Prove neg_neg : $(-(-x)) = x$
= inv_inv x

▷ Prove neg_plus : $((-x) + x) = \mathbf{0}$
= inv_o x

▷ Prove $\mathsf{ABGROUP}_1$: $(x = y) \leftrightarrow ((x - y) = \mathbf{0})$
= $\mathsf{GROUP}_1\ x\ y$

▷ Prove minus_neg : $(x - (-y)) = (x + y)$
= $\mathsf{o}_2\ x\ y$.inv_inv

▷ Prove subresult Q_1 : $(-(x + y)) = ((-y) + (-x))$
= inv_of_o $x\ y$

▷ Prove subresult Q_2 : $((-y) + (-x)) = ((-x) + (-y))$
= plus_comm $(-y)\ (-x)$

▷ Prove neg_of_plus : $(-(x + y)) = ((-x) + (-y))$
= Q_1.tran Q_2

▷ Discharge $\mathsf{Q}_2, \mathsf{Q}_1$

▷ Prove subresult Q_1 : $(-(x - y)) = ((-(-y)) - x)$
= $\mathsf{inv\_of\_o}\ x\ (-y)$

▷ Prove subresult Q_2 : $(y^{-1-1} - x) = (y - x)$
= $y.\mathsf{inv\_inv.minus}_1\ x$

▷ Prove $\mathsf{neg\_of\_minus}$: $(-(x - y)) = (y - x)$
= $\mathsf{Q}_1.\mathsf{tran}\ \mathsf{Q}_2$

▷ Discharge $\mathsf{Q}_2, \mathsf{Q}_1$

▷ Prove subresult Q_1 : $((w + x) + (y + z)) = (w + (x + (y + z)))$
= $\mathsf{plus\_assoc}\ w\ x\ (y + z)$

▷ Prove subresult $\mathsf{Q}_{2\text{-}1}$: $(x + (y + z)) = ((x + y) + z)$
= $(\mathsf{plus\_assoc}\ x\ y\ z).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_{2\text{-}2}$: $((x + y) + z) = ((y + x) + z)$
= $(\mathsf{plus\_comm}\ x\ y).\mathsf{plus}_1\ z$

▷ Prove subresult $\mathsf{Q}_{2\text{-}3}$: $((y + x) + z) = (y + (x + z))$
= $\mathsf{plus\_assoc}\ y\ x\ z$

▷ Prove subresult Q_2 : $(w + (x + (y + z))) = (w + (y + (x + z)))$
= $w.\mathsf{plus}_2\ (\mathsf{Q}_{2\text{-}1}.\mathsf{tran}\ (\mathsf{Q}_{2\text{-}2}.\mathsf{tran}\ \mathsf{Q}_{2\text{-}3}))$

▷ Prove subresult Q_3 : $(w + (y + (x + z))) = ((w + y) + (x + z))$
= $(\mathsf{plus\_assoc}\ w\ y\ (x + z)).\mathsf{symm}$

▷ Prove $\mathsf{ABGROUP}_2$: $((w + x) + (y + z)) = ((w + y) + (x + z))$
= $\mathsf{Q}_1.\mathsf{tran}\ (\mathsf{Q}_2.\mathsf{tran}\ \mathsf{Q}_3)$

▷ Discharge $\mathsf{Q}_3, \mathsf{Q}_2, \mathsf{Q}_{2\text{-}3}, \mathsf{Q}_{2\text{-}2}, \mathsf{Q}_{2\text{-}1}, \mathsf{Q}_1$

▷ Discharge but keep z, y, x, w

▷ Prove subresult Q_1 : $((w + x) + (-(y + z))) = ((w + x) + ((-y) + (-z)))$
= $(w + x).\mathsf{plus}_2\ (\mathsf{neg\_of\_plus}\ y\ z)$

▷ Prove subresult Q_2
: $((w + x) + ((-y) + (-z))) = ((w + (-y)) + (x + (-z)))$
= $\mathsf{ABGROUP}_2\ w\ x\ (-y)\ (-z)$

▷ Prove $\mathsf{ABGROUP}_3$: $((w + x) - (y + z)) = ((w - y) + (x - z))$
= $\mathsf{Q}_1.\mathsf{tran}\ \mathsf{Q}_2$

▷ Discharge $\mathsf{Q}_2, \mathsf{Q}_1, z, y, x, w, G$

B.5.1.5 Fields

▷ Define $\mathsf{field\_axioms} = [\lambda F \,|\, \mathsf{abelian\_group}]\ [\lambda un : F]\ [\lambda times : \mathsf{map}_2\ F\ F\ F]$

$[\lambda recip : \mathsf{map}\ F\ F]\ (\bigwedge_3\ (\{\forall x, y : F\}\ (x.times\ y) = (y.times\ x))$

$(\{\forall x, y, z : F\}\ ((x.times\ y).times\ z) = (x.times\ (y.times\ z)))$

$(\{\forall x, y, z : F\}\ (x.times\ (y + z)) = ((x.times\ y) + (x.times\ z)))) \wedge$

$(\bigwedge_3\ (un \neq \mathbf{0})\ (\{\forall x : F\}\ (x.times\ un) = x)$

$(\{\forall x : F\}\ ((x = \mathbf{0}) \wedge ((recip\ x) = \mathbf{0})) \vee ((x.times\ (recip\ x)) = un)))$

$: \{\Pi F \,|\, \mathsf{abelian\_group}\}\ F \to (\mathsf{map}_2\ F\ F\ F) \to (\mathsf{map}\ F\ F) \to \mathsf{prop}$

▷ Define $\mathsf{field} = \langle \Sigma F : \mathsf{abelian\_group} \rangle\ \langle \Sigma un : F \rangle\ \langle \Sigma times : \mathsf{map}_2\ F\ F\ F \rangle$

$\langle \Sigma recip : \mathsf{map}\ F\ F \rangle\ \mathsf{field\_axioms}\ un\ times\ recip : \mathsf{Type}_1$

▷ Unless otherwise specified, by default $F \mid \mathsf{field}$

▷ Define coercion $\mathsf{fg} = [\lambda F : \mathsf{field}]\ F_{.\mathbf{1}} : \mathsf{field} \to \mathsf{abelian\_group}$

▷ Allow $\times$ to be written infix

▷ Allow ${}^1\!/$ to be written prefix

▷ Introduce F

▷ Define $\mathbf{1} = F_{.\mathbf{2}\cdot\mathbf{1}} : F$

▷ Define $\times = F_{.\mathbf{2}\cdot\mathbf{2}\cdot\mathbf{1}} : \mathsf{map}_2\ F\ F\ F$

▷ Define ${}^1\!/ = F_{.\mathbf{2}\cdot\mathbf{2}\cdot\mathbf{2}\cdot\mathbf{1}} : \mathsf{map}\ F\ F$

▷ Introduce $x_1, x_2 \mid F$; suppose $Qx : x_1 = x_2$; introduce $x, y, z : F$; $y_1, y_2 \mid F$ and suppose $Qy : y_1 = y_2$

▷ Prove $\mathsf{times\_resp} : (x_1 \times y_1) = (x_2 \times y_2)$
$= \mathsf{resp}_2\ \times\ Qx\ Qy$

▷ Prove $\mathsf{times}_1 : (x_1 \times y) = (x_2 \times y)$
$= \mathsf{resps}_1\ \times\ Qx\ y$

▷ Prove $\mathsf{times}_2 : (x \times y_1) = (x \times y_2)$
$= \mathsf{resps}_2\ \times\ x\ Qy$

▷ Prove $\mathsf{recip\_resp} : ({}^1\!/x_1) = ({}^1\!/x_2)$
$= \mathsf{resp}\ {}^1\!/\ Qx$

▷ Prove recip$_1$: $({}^1/x_1) = ({}^1/x_2)$
= recip_resp

▷ Prove times_comm : $(x \times y) = (y \times x)$
= $F_{.\mathbf{2}.\mathbf{2}.\mathbf{2}.\mathbf{2}}$.fst.fst$_3$ x y

▷ Prove times_assoc : $((x \times y) \times z) = (x \times (y \times z))$
= $F_{.\mathbf{2}.\mathbf{2}.\mathbf{2}.\mathbf{2}}$.fst.snd$_3$ x y z

▷ Prove times_plus : $(x \times (y + z)) = ((x \times y) + (x \times z))$
= $F_{.\mathbf{2}.\mathbf{2}.\mathbf{2}.\mathbf{2}}$.fst.thd$_3$ x y z

▷ Prove nontriv : $\mathbf{1} \neq \mathbf{0}$
= $F_{.\mathbf{2}.\mathbf{2}.\mathbf{2}.\mathbf{2}}$.snd.fst$_3$

▷ Prove times_un : $(x \times \mathbf{1}) = x$
= $F_{.\mathbf{2}.\mathbf{2}.\mathbf{2}.\mathbf{2}}$.snd.snd$_3$ x

▷ Prove times_recip : $((x = \mathbf{0}) \wedge (({}^1/x) = \mathbf{0})) \vee ((x \times ({}^1/x)) = \mathbf{1})$
= $F_{.\mathbf{2}.\mathbf{2}.\mathbf{2}.\mathbf{2}}$.snd.thd$_3$ x

▷ Discharge $Qy, y_2, y_1, z, y, x, Qx, x_2, x_1, F$

B.5.1.6 Results concerning fields

▷ Introduce F and $x, y, z : F$

▷ Prove un_times : $(\mathbf{1} \times x) = x$
= tran (times_comm **1** x) (times_un x)

▷ Prove subresult P$_1$: $((x + y) \times z) = (z \times (x + y))$
= times_comm $(x + y)$ z

▷ Prove subresult P$_2$: $(z \times (x + y)) = ((z \times x) + (z \times y))$
= times_plus z x y

▷ Prove subresult P$_3$: $((z \times x) + (z \times y)) = ((x \times z) + (y \times z))$
= plus_resp (times_comm z x) (times_comm z y)

▷ Prove plus_times : $((x + y) \times z) = ((x \times z) + (y \times z))$
= P$_1$.tran (P$_2$.tran P$_3$)

▷ Discharge P$_3$, P$_2$, P$_1$

▷ Discharge but keep z, y, x

▷ Let zx $= \mathbf{0} \times x : F_{.\mathbf{1}.\mathbf{1}.\mathbf{1}}$

▷ Prove subresult P$_1$: zx = (zx + $\mathbf{0}$)
= zx.plus_ze.symm

▷ Prove subresult P_2 : $(\mathsf{zx} + \mathbf{0}) = (\mathsf{zx} + (\mathsf{zx} - \mathsf{zx}))$
= zx.plus$_2$ zx.minus_self.symm

▷ Prove subresult P_3 : $(\mathsf{zx} + (\mathsf{zx} + (-\mathsf{zx}))) = ((\mathsf{zx} + \mathsf{zx}) + (-\mathsf{zx}))$
= (plus_assoc zx zx $(-\mathsf{zx})$).symm

▷ Prove subresult $\mathsf{P}_{4\text{-}1}$: $(\mathsf{zx} + \mathsf{zx}) = ((\mathbf{0} + \mathbf{0}) \times x)$
= (plus_times $\mathbf{0}$ $\mathbf{0}$ x).symm

▷ Prove subresult $\mathsf{P}_{4\text{-}2}$: $((\mathbf{0} + \mathbf{0}) \times x) = \mathsf{zx}$
= (ze_plus $\mathbf{0}$).times$_1$ x

▷ $\mathsf{P}_{4\text{-}1}$.tran $\mathsf{P}_{4\text{-}2}$: $(\mathsf{zx} + \mathsf{zx}) = \mathsf{zx}$

▷ Prove subresult P_4 : $((\mathsf{zx} + \mathsf{zx}) - \mathsf{zx}) = (\mathsf{zx} - \mathsf{zx})$
= ($\mathsf{P}_{4\text{-}1}$.tran $\mathsf{P}_{4\text{-}2}$).minus$_1$ zx

▷ Prove subresult P_5 : $(\mathsf{zx} - \mathsf{zx}) = \mathbf{0}$
= minus_self zx

▷ Prove ze_times : $(\mathbf{0} \times x) = \mathbf{0}$
= P_1.tran (P_2.tran (P_3.tran (P_4.tran P_5)))

▷ Discharge $\mathsf{P}_5, \mathsf{P}_4, \mathsf{P}_{4\text{-}2}, \mathsf{P}_{4\text{-}1}, \mathsf{P}_3, \mathsf{P}_2, \mathsf{P}_1, \mathsf{zx}$

▷ Prove times_ze : $(x \times \mathbf{0}) = \mathbf{0}$
= (times_comm x $\mathbf{0}$).tran ze_times

▷ Discharge but keep z, y, x

▷ Suppose H_1 : $(x = \mathbf{0}) \wedge ({}^{1}\!/\!_{x} = \mathbf{0})$

▷ Prove subresult P_1 : $x = \mathbf{0}$
= H_1.fst

▷ Prove subresult C_1 : $(x = \mathbf{0})$.or_not
= or_not_is_true P_1

▷ Suppose H_2 : $(x \times ({}^{1}\!/\!_{x})) = \mathbf{1}$

▷ Suppose $H_{2\text{-}1}$: $x = \mathbf{0}$

▷ Prove subresult $\mathsf{P}_{2\text{-}1}$: $\mathbf{1} = (x \times ({}^{1}\!/\!_{x}))$
= H_2.symm

▷ Prove subresult $\mathsf{P}_{2\text{-}2}$: $(x \times ({}^{1}\!/\!_{x})) = (\mathbf{0} \times ({}^{1}\!/\!_{x}))$
= $H_{2\text{-}1}$.times$_1$ $({}^{1}\!/\!_{x})$

▷ Prove subresult $\mathsf{P}_{2\text{-}3}$: $(\mathbf{0} \times ({}^{1}\!/\!_{x})) = \mathbf{0}$
= ze_times $({}^{1}\!/\!_{x})$

▷ $\mathsf{P}_{2\text{-}1}$.tran ($\mathsf{P}_{2\text{-}2}$.tran $\mathsf{P}_{2\text{-}3}$) : $\mathbf{1} = \mathbf{0}$

▷ Prove subresult P_2 **:** false
= nontriv ($\mathsf{P}_{2\text{-}1}$.tran ($\mathsf{P}_{2\text{-}2}$.tran $\mathsf{P}_{2\text{-}3}$))

▷ Discharge $H_{2\text{-}1}$

▷ P_2 **:** $x \neq \mathbf{0}$

▷ Prove subresult C_2 **:** $(x = \mathbf{0})$.or_not
= or_not_is_false P_2

▷ Discharge H_2, H_1

▷ Prove subresult ze_non_ze **:** $(x = \mathbf{0})$.or_not
= case (times_recip x) C_1 C_2

▷ Discharge z, y, x

▷ Prove FIELD_1 **:** F.is_discrete
= $[\lambda x, y : F]$ DEC_2 ($\mathsf{ABGROUP}_1$ x y).iff_symm (ze_non_ze $(x - y)$)

▷ Discharge ze_non_ze, $\mathsf{C}_2, \mathsf{P}_2, \mathsf{P}_{2\text{-}3}, \mathsf{P}_{2\text{-}2}, \mathsf{P}_{2\text{-}1}, \mathsf{C}_1, \mathsf{P}_1$

▷ Suppose H **:** $(\mathbf{0} \times (^1/\mathbf{0})) = \mathbf{1}$

▷ Prove subresult $\mathsf{P}_{2\text{-}1}$ **:** $(\mathbf{0} \times (^1/\mathbf{0})) = \mathbf{0}$
= ze_times $(^1/\mathbf{0})$

▷ H.symm.tran $\mathsf{P}_{2\text{-}1}$ **:** $\mathbf{1} = \mathbf{0}$

▷ Prove subresult P_2 **:** false
= nontriv (H.symm.tran $\mathsf{P}_{2\text{-}1}$)

▷ Prove subresult C_2 **:** $(^1/\mathbf{0}) = \mathbf{0}$
= ex_falso $((^1/\mathbf{0}) = \mathbf{0})$ P_2

▷ Discharge H

▷ Prove ze_recip **:** $(^1/\mathbf{0}) = \mathbf{0}$
= case (times_recip $\mathbf{0}$) snd C_2

▷ Discharge $\mathsf{C}_2, \mathsf{P}_2, \mathsf{P}_{2\text{-}1}, F$

B.5.2 Substructures

B.5.2.1 Subgroups

▷ Define $\mathsf{subgroup\_axioms} = [\lambda G \,|\, \mathsf{group}]\ [\lambda U : \mathsf{subset}\ G]$

$\bigwedge_3\ (\mathsf{id} \in U)\ (\{\forall x, y : U\}\ (x \circ y) \in U)\ (\{\forall x : U\}\ x^{-1} \in U)$

$: \{\Pi G \,|\, \mathsf{group}\}\ (\mathsf{subset}\ G) \to \mathsf{prop}$

▷ Introduce G

▷ Define $\mathsf{subgroup} = \langle \Sigma U : \mathsf{subset}\ G \rangle\ \mathsf{subgroup\_axioms}\ U : \mathsf{Type}_1$

▷ Define coercion $\mathsf{subcar} = [\lambda U : \mathsf{subgroup}]\ U_{.\mathbf{1}} : \mathsf{subgroup} \to \mathsf{subset}\ G$

▷ Discharge G

▷ Introduce $G \mid \mathsf{group}$

▷ Unless otherwise specified, by default $D, U : \mathsf{subgroup}\ G$

▷ Define $\mathsf{subgroup\_of} = [\lambda U]\ \langle \Sigma D \rangle\ D \subseteq U : (\mathsf{subgroup}\ G) \to \mathsf{Type}_1$

▷ Introduce $U \mid \mathsf{subgroup}\ G$ and $x, y : U$

▷ Prove $\mathsf{id\_closed} : \mathsf{id} \in U$
$= U_{.\mathbf{2}}.\mathsf{fst}_3$

▷ Prove $\mathsf{o\_closed} : (x \circ y) \in U$
$= U_{.\mathbf{2}}.\mathsf{snd}_3\ x\ y$

▷ Prove $\mathsf{inv\_closed} : x^{-1} \in U$
$= U_{.\mathbf{2}}.\mathsf{thd}_3\ x$

▷ Define coercion $\mathsf{unsubg} = [\lambda D : \mathsf{subgroup\_of}\ U]\ D_{.\mathbf{1}}$

$: (\mathsf{subgroup\_of}\ U) \to \mathsf{subgroup}\ G$

▷ Define coercion $\mathsf{make\_subg} = [\lambda D \,|\, \mathsf{subgroup}\ G]\ [\lambda H : D \subseteq U]$

$(D, H : \mathsf{subgroup\_of}\ U) : \{\Pi D \,|\, \mathsf{subgroup}\ G\}\ (D \subseteq U) \to \mathsf{subgroup\_of}\ U$

▷ Discharge y, x

▷ Introduce $x_1, x_2 \mid U$; suppose $Qx : x_1 = x_2$; introduce $y_1, y_2 \mid U$ and suppose $Qy : y_1 = y_2$

▷ Prove $\mathsf{subgroup\_o\_forms\_map}_2 : (x_1 \circ y_1) = (x_2 \circ y_2)$
$= \circ.\mathsf{resp}_2\ Qx\ Qy$

▷ Prove subgroup_inv_forms_map : ${x_1}^{-1} = {x_2}^{-1}$
= $^{-1}$.resp Qx

▷ Discharge $Qy, y_2, y_1, Qx, x_2, x_1$

▷ Define subgroup_id = (id$|G$, id_closed : U) : U

▷ Define subgroup_o = ($[\lambda x, y : U]$ $(x \circ y, x$.o_closed $y : U)$,
subgroup_o_forms_map$_2$: map$_2$ U U U) : map$_2$ U U U

▷ Define subgroup_inv
= ($[\lambda x : U]$ $(x^{-1}, x$.inv_closed : $U)$, subgroup_inv_forms_map : map U U)
: map U U

▷ Introduce x, y, z : U

▷ Prove subresult subgroup_o_assoc : $((x \circ y) \circ z) = (x \circ (y \circ z))$
= o_assoc x y z

▷ Prove subresult subgroup_o_id : $(x \circ$ id$) = x$
= o_id x

▷ Prove subresult subgroup_id_o : (id $\circ x) = x$
= id_o x

▷ Prove subresult subgroup_o_inv : $(x \circ x^{-1}) =$ id
= o_inv x

▷ Discharge z, y, x

▷ Prove subgroup_forms_group
: group_axioms$|U$ subgroup_id subgroup_o subgroup_inv
= pair$_3$ subgroup_o_assoc (subgroup_o_id.pair subgroup_id_o) subgroup_o_inv

▷ Discharge subgroup_o_inv, subgroup_id_o, subgroup_o_id, subgroup_o_assoc, U

▷ Define coercion as_a_group = $[\lambda U]$ (U, subgroup_id$|U$, subgroup_o$|U$,
subgroup_inv$|U$, subgroup_forms_group$|U$: group) : (subgroup G) $\to$ group

▷ Introduce U_1, U_2 : subgroup G and x, y : $U_1 \cap U_2$

▷ Prove subresult intersect_id_closed : (id $\in U_1$) $\wedge$ (id $\in U_2$)
= pair (id_closed$|U_1$) (id_closed$|U_2$)

▷ Prove subresult intersect_o_closed
: $((x$.ev.fst $\circ\ y$.ev.fst$) \in U_1) \wedge ((x$.ev.snd $\circ\ y$.ev.snd$) \in U_2)$
= pair (x.ev.fst.o_closed y.ev.fst) (x.ev.snd.o_closed y.ev.snd)

▷ Prove subresult intersect_inv_closed : $(x.\mathsf{ev.fst}^{-1} \in U_1) \wedge (x.\mathsf{ev.snd}^{-1} \in U_2)$
$= \mathsf{pair}\ x.\mathsf{ev.fst.inv\_closed}\ x.\mathsf{ev.snd.inv\_closed}$

▷ Discharge y, x

▷ Prove intersect_forms_subgroup : subgroup_axioms $(U_1 \cap U_2)$
$= \mathsf{pair}_3\ \mathsf{intersect\_id\_closed}\ \mathsf{intersect\_o\_closed}\ \mathsf{intersect\_inv\_closed}$

▷ Explicitly overload the identifier $\cap$

▷ Define $\cap =$ $(U_1 \cap U_2, \mathsf{intersect\_forms\_subgroup} : \mathsf{subgroup}\ G)$: subgroup G

▷ Discharge intersect_inv_closed, intersect_o_closed, intersect_id_closed, U_2, U_1, G

B.5.2.2 Subgroups of abelian groups

▷ Introduce G | abelian_group

▷ Prove subgroup_forms_abelian_group
: $\{\forall H : \mathsf{subgroup}\ G\}$ abelian_group_axioms H
$= [\lambda H : \mathsf{subgroup}\ G]\ [\lambda x, y : H]\ \mathsf{plus\_comm}\ x\ y$

▷ Define coercion as_an_abgroup

$= [\lambda H : \mathsf{subgroup}\ G]\ (H, \mathsf{subgroup\_forms\_abelian\_group}\ H : \mathsf{abelian\_group})$

: $(\mathsf{subgroup}\ G) \to \mathsf{abelian\_group}$

▷ Introduce H | subgroup G and $x, y : H$

▷ Prove ze_closed : $\mathbf{0} \in H$
$= \mathsf{id\_closed}|G|H$

▷ Prove plus_closed : $(x + y) \in H$
$= \mathsf{o\_closed}|G\ x\ y$

▷ Prove neg_closed : $(-x) \in H$
$= \mathsf{inv\_closed}|G\ x$

▷ Prove minus_closed : $(x - y) \in H$
$= \mathsf{o\_closed}|G\ x\ (\mathsf{inv\_closed}|G\ y)$

▷ Discharge y, x, H, G

B.5.2.3 Subfields

▷ Define $\mathsf{subfield\_axioms} = [\lambda F]\ [\lambda K : \mathsf{subgroup}\ F]$

$\bigwedge_3\ (\mathbf{1} \in K)\ (\{\forall x, y : K\}\ (x \times y) \in K)\ (\{\forall x : K\}\ ({}^1\!/x) \in K)$

$: \{\Pi F\}\ (\mathsf{subgroup}\ F) \to \mathsf{prop}$

▷ Define $\mathsf{subfield} = [\lambda F : \mathsf{field}]\ \langle \Sigma K : \mathsf{subgroup}\ F \rangle\ \mathsf{subfield\_axioms}\ K$

$: \mathsf{field} \to \mathsf{Type}_1$

▷ Define coercion $\mathsf{subfg} = [\lambda F]\ [\lambda K]\ K_{.\mathbf{1}} : \{\Pi F\}\ (\mathsf{subfield}\ F) \to \mathsf{subgroup}\ F$

▷ Introduce F

▷ Unless otherwise specified, by default $K, K_1, K_2, L, L_1, L_2 : \mathsf{subfield}\ F$

▷ Define $\mathsf{subfield\_of} = [\lambda L]\ \langle \Sigma K \rangle\ K \subseteq L : (\mathsf{subfield}\ F) \to \mathsf{Type}_1$

▷ Introduce $L \mid \mathsf{subfield}\ F$ and $x, y : L$

▷ Prove $\mathsf{un\_closed} : \mathbf{1} \in L$

$= L_{.\mathbf{2}}.\mathsf{fst}_3$

▷ Prove $\mathsf{times\_closed} : (x \times y) \in L$

$= L_{.\mathbf{2}}.\mathsf{snd}_3\ x\ y$

▷ Prove $\mathsf{recip\_closed} : ({}^1\!/x) \in L$

$= L_{.\mathbf{2}}.\mathsf{thd}_3\ x$

▷ Define coercion $\mathsf{unsubf} = [\lambda K : \mathsf{subfield\_of}\ L]\ K_{.\mathbf{1}}$

$: (\mathsf{subfield\_of}\ L) \to \mathsf{subfield}\ F$

▷ Define coercion $\mathsf{make\_subf} = [\lambda K \mid \mathsf{subfield}\ F]\ [\lambda H : K \subseteq L]$

$(K, H : \mathsf{subfield\_of}\ L) : \{\Pi K \mid \mathsf{subfield}\ F\}\ (K \subseteq L) \to \mathsf{subfield\_of}\ L$

▷ Discharge y, x

▷ Introduce $x_1, x_2 \mid L$; suppose $Qx : x_1 = x_2$; introduce $y_1, y_2 \mid L$ and

suppose $Qy : y_1 = y_2$

▷ Prove $\mathsf{subfield\_times\_forms\_map}_2 : (x_1 \times y_1) = (x_2 \times y_2)$

$= \times.\mathsf{resp}_2\ Qx\ Qy$

▷ Prove $\mathsf{subfield\_recip\_forms\_map} : ({}^1\!/x_1) = ({}^1\!/x_2)$

$= \mathsf{recip\_resp}\ Qx$

▷ Discharge $Qy, y_2, y_1, Qx, x_2, x_1$

▷ Define $\mathsf{subfield\_un} = (\mathbf{1}|F, \mathsf{un\_closed} : L) : L$

▷ Define $\mathsf{subfield\_times} = ([\lambda x, y : L]\ (x \times y, x.\mathsf{times\_closed}\ y : L),$

$\mathsf{subfield\_times\_forms\_map}_2 : \mathsf{map}_2\ L\ L\ L) : \mathsf{map}_2\ L\ L\ L$

▷ Define $\mathsf{subfield\_recip}$

$= ([\lambda x : L]\ ({}^1\!/x, \mathsf{recip\_closed}\ x : L), \mathsf{subfield\_recip\_forms\_map} : \mathsf{map}\ L\ L)$

$: \mathsf{map}\ L\ L$

▷ Introduce $x, y, z : L$

▷ Prove subresult $\mathsf{subfield\_times\_comm} : (x \times y) = (y \times x)$

$= \mathsf{times\_comm}\ x\ y$

▷ Prove subresult $\mathsf{subfield\_times\_assoc} : ((x \times y) \times z) = (x \times (y \times z))$

$= \mathsf{times\_assoc}\ x\ y\ z$

▷ Prove subresult $\mathsf{subfield\_times\_plus} : (x \times (y + z)) = ((x \times y) + (x \times z))$

$= \mathsf{times\_plus}\ x\ y\ z$

▷ Prove subresult $\mathsf{subfield\_times\_un} : (x \times \mathbf{1}) = x$

$= \mathsf{times\_un}\ x$

▷ Prove subresult $\mathsf{subfield\_times\_recip}$

$: ((x = \mathbf{0}) \wedge (({}^1\!/x) = \mathbf{0})) \vee ((x \times ({}^1\!/x)) = \mathbf{1})$

$= \mathsf{times\_recip}\ x$

▷ Discharge z, y, x

▷ Prove $\mathsf{subfield\_forms\_field}$

$: \mathsf{field\_axioms}|L\ \mathsf{subfield\_un}\ \mathsf{subfield\_times}\ \mathsf{subfield\_recip}$

$= \mathsf{pair}\ (\mathsf{pair}_3\ \mathsf{subfield\_times\_comm}\ \mathsf{subfield\_times\_assoc}\ \mathsf{subfield\_times\_plus})$

$(\mathsf{pair}_3\ (\mathsf{nontriv}|F)\ \mathsf{subfield\_times\_un}\ \mathsf{subfield\_times\_recip})$

▷ Discharge $\mathsf{subfield\_times\_recip}, \mathsf{subfield\_times\_un}, \mathsf{subfield\_times\_plus},$

$\mathsf{subfield\_times\_assoc}, \mathsf{subfield\_times\_comm}, L$

▷ Define coercion $\mathsf{as\_a\_field} = [\lambda L]\ (L, \mathsf{subfield\_un}|L, \mathsf{subfield\_times}|L,$

$\mathsf{subfield\_recip}|L, \mathsf{subfield\_forms\_field}|L : \mathsf{field}) : (\mathsf{subfield}\ F) \to \mathsf{field}$

▷ Introduce K, L and $x, y : K \cap L$

▷ Prove subresult $\mathsf{intersect\_un\_closed} : (\mathbf{1} \in K) \wedge (\mathbf{1} \in L)$

$= \mathsf{pair}\ (\mathsf{un\_closed}|K)\ (\mathsf{un\_closed}|L)$

▷ Prove subresult $\mathsf{intersect\_times\_closed}$

$: ((x.\mathsf{ev.fst} \times y.\mathsf{ev.fst}) \in K) \wedge ((x.\mathsf{ev.snd} \times y.\mathsf{ev.snd}) \in L)$

$= \mathsf{pair}\ (x.\mathsf{ev.fst.times\_closed}\ y.\mathsf{ev.fst})\ (x.\mathsf{ev.snd.times\_closed}\ y.\mathsf{ev.snd})$

▷ Prove subresult intersect_recip_closed : $((^1/x.\mathsf{ev.fst}) \in K) \wedge ((^1/x.\mathsf{ev.snd}) \in L)$
= pair (recip_closed x.ev.fst) (recip_closed x.ev.snd)

▷ Discharge y, x

▷ Prove intersect_forms_subfield : subfield_axioms $(K \cap L)$
= pair_3 intersect_un_closed intersect_times_closed intersect_recip_closed

▷ Explicitly overload the identifier $\cap$

▷ Define $\cap$ = $(K \cap L, \mathsf{intersect\_forms\_subfield} : \mathsf{subfield}\ F) : \mathsf{subfield}\ F$

▷ Discharge

intersect_recip_closed, intersect_times_closed, intersect_un_closed, L, K, F

B.5.3 Morphisms between substructures

B.5.3.1 Morphisms that respect binary mappings

▷ Explicitly overload the identifier $\oplus$

▷ Allow $\oplus_1$ to be written infix

▷ Allow $\oplus_2$ to be written infix

▷ Introduce S, T; $\oplus_1 : \mathsf{map}_2\ S\ S\ S$; $\oplus_2 : \mathsf{map}_2\ T\ T\ T$ and $A : \mathsf{subset}\ S$

▷ Define $\mathsf{is\_resp\_map}_2 = [\lambda f : \mathsf{map}\ S\ T]\ \{\forall x, y : A\}$

$(f\ (x \oplus_1 y)) = ((f\ x) \oplus_2 (f\ y)) : (\mathsf{map}\ S\ T) \to \mathsf{prop}$

▷ Introduce $f_1, f_2 \mid \mathsf{map}\ S\ T$; $Q : f_1 = f_2$; $H : f_1.\mathsf{is\_resp\_map}_2$ and $x, y : A$

▷ Prove subresult $\mathsf{Q}_1 : (f_{2\cdot\mathbf{1}}\ (x \oplus_1 y)) = (f_{1\cdot\mathbf{1}}\ (x \oplus_1 y))$
= $(Q\ (x \oplus_1 y)).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_2 : (f_1\ (x \oplus_1 y)) = ((f_1\ x) \oplus_2 (f_1\ y))$
= $H\ x\ y$

▷ Prove subresult $\mathsf{Q}_3 : (\oplus_2\ (f_{1\cdot\mathbf{1}}\ x)\ (f_{1\cdot\mathbf{1}}\ y)) = (\oplus_2\ (f_{2\cdot\mathbf{1}}\ x)\ (f_{2\cdot\mathbf{1}}\ y))$
= $\oplus_2.\mathsf{resp}_2\ (Q\ x)\ (Q\ y)$

▷ Prove resp_map$_2$_forms_subset : $(f_{2\cdot\mathbf{1}}\ (x \oplus_1 y)) = (\oplus_2\ (f_{2\cdot\mathbf{1}}\ x)\ (f_{2\cdot\mathbf{1}}\ y))$
= $\mathsf{Q}_1.\mathsf{tran}\ (\mathsf{Q}_2.\mathsf{tran}\ \mathsf{Q}_3)$

▷ Discharge $\mathsf{Q}_3, \mathsf{Q}_2, \mathsf{Q}_1, y, x, H, Q, f_2, f_1$

▷ Define $\mathsf{resp\_map}_2$

$= (\mathsf{is\_resp\_map}_2, \mathsf{resp\_map}_2\mathsf{\_forms\_subset} : \mathsf{subset}\ (\mathsf{map}\ S\ T))$

$: \mathsf{subset}\ (\mathsf{map}\ S\ T)$

▷ Discharge $A, \oplus_2, \oplus_1, T, S$

B.5.3.2 Morphisms between subgroups

▷ Introduce $G \mid \mathsf{group}$

▷ Define $\mathsf{is\_subgroup\_hom} = [\lambda U : \mathsf{subset}\ G]\ \mathsf{is\_resp\_map}_2|G|G \circ \circ U$

$: (\mathsf{subset}\ G) \to (\mathsf{map}\ G\ G) \to \mathsf{prop}$

▷ Define $\mathsf{subgroup\_hom} = [\lambda U : \mathsf{subset}\ G]\ \mathsf{resp\_map}_2|G|G \circ \circ U$

$: (\mathsf{subset}\ G) \to \mathsf{subset}\ (\mathsf{map}\ G\ G)$

▷ Introduce $U \mid \mathsf{subgroup}\ G$; $f : \mathsf{subgroup\_hom}\ U$ and $x, y : U$

▷ Prove $\mathsf{on\_o} : (f\ (x \circ y)) = ((f\ x) \circ (f\ y))$

$= f.\mathsf{ev}\ x\ y$

▷ Prove subresult $\mathsf{Q}_1 : ((f\ (\mathsf{id}|U)) \circ (f\ (\mathsf{id}|U))) = (f\ ((\mathsf{id}|U) \circ (\mathsf{id}|U)))$

$= (f.\mathsf{ev}\ (\mathsf{id}|U)\ (\mathsf{id}|U)).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_2 : (f\ (\mathsf{id} \circ \mathsf{id})) = (f\ \mathsf{id})$

$= f.\mathsf{resp}\ \mathsf{id.o\_id}$

▷ Prove $\mathsf{on\_id} : (f\ \mathsf{id}) = \mathsf{id}$

$= \mathsf{GROUP}_4\ (\mathsf{Q}_1.\mathsf{tran}\ \mathsf{Q}_2)$

▷ Discharge $\mathsf{Q}_2, \mathsf{Q}_1$

▷ Prove subresult $\mathsf{Q}_1 : (f\ x^{-1}) = ((f\ x^{-1}) \circ \mathsf{id})$

$= (f\ x^{-1}).\mathsf{o\_id.symm}$

▷ Prove subresult $\mathsf{Q}_2 : ((f\ x^{-1}) \circ \mathsf{id}) = ((f\ x^{-1}) \circ ((f\ x) \circ (f\ x)^{-1}))$

$= (f\ x^{-1}).\mathsf{o}_2\ (f\ x).\mathsf{o\_inv.symm}$

▷ Prove subresult Q_3

$: ((f\ x^{-1}) \circ ((f\ x) \circ (f\ x)^{-1})) = (((f\ x^{-1}) \circ (f\ x)) \circ (f\ x)^{-1})$

$= (\mathsf{o\_assoc}\ (f\ x^{-1})\ (f\ x)\ (f\ x)^{-1}).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_{4\_1} : ((f\ x(^{-1}|U)) \circ (f\ x)) = (f\ (x(^{-1}|U) \circ x))$

$= (f.\mathsf{ev}\ x(^{-1}|U)\ x).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_{4\_2} : (f\ (x^{-1} \circ x)) = (f\ \mathsf{id})$

$= f.\mathsf{resp}\ (\mathsf{inv\_o}\ x)$

▷ Prove subresult Q_4 : $(((f\ x(^{-1}|U)) \circ (f\ x)) \circ (f\ x)^{-1}) = (\mathsf{id} \circ (f\ x)^{-1})$
= $(\mathsf{Q}_{4\text{-}1}.\mathsf{tran}\ (\mathsf{Q}_{4\text{-}2}.\mathsf{tran}\ \mathsf{on\_id})).\mathsf{o}_1\ (f\ x)^{-1}$

▷ Prove subresult Q_5 : $(\mathsf{id} \circ (f\ x)^{-1}) = (f\ x)^{-1}$
= $\mathsf{id\_o}\ (f\ x)^{-1}$

▷ Prove on_inv : $(f\ x^{-1}) = (f\ x)^{-1}$
= $\mathsf{Q}_1.\mathsf{tran}\ (\mathsf{Q}_2.\mathsf{tran}\ (\mathsf{Q}_3.\mathsf{tran}\ (\mathsf{Q}_4.\mathsf{tran}\ \mathsf{Q}_5)))$

▷ Discharge $\mathsf{Q}_5, \mathsf{Q}_4, \mathsf{Q}_{4\text{-}2}, \mathsf{Q}_{4\text{-}1}, \mathsf{Q}_3, \mathsf{Q}_2, \mathsf{Q}_1, y, x, f, U, G$

▷ Introduce G | abelian_group; H | subgroup G; f : subgroup_hom H and x, y : H

▷ Prove on_plus : $(f\ (x + y)) = ((f\ x) + (f\ y))$
= $f.\mathsf{on\_o}\ x\ y$

▷ Prove on_ze : $(f\ \mathbf{0}) = \mathbf{0}$
= $f.\mathsf{on\_id}$

▷ Prove on_neg : $(f\ (-x)) = (-(f\ x))$
= $f.\mathsf{on\_inv}\ x$

▷ Discharge but keep y, x

▷ Prove subresult P_1 : $(f\ (x + (-|Hy))) = ((f\ x) + (f\ (-|Hy)))$
= $\mathsf{on\_plus}\ x\ (-|Hy)$

▷ Prove subresult P_2 : $((f\ x) + (f\ (-y))) = ((f\ x) + (-(f\ y)))$
= $(f\ x).\mathsf{plus}_2\ (\mathsf{on\_neg}\ y)$

▷ Prove on_minus : $(f\ (x - y)) = ((f\ x) - (f\ y))$
= $\mathsf{P}_1.\mathsf{tran}\ \mathsf{P}_2$

▷ Discharge $\mathsf{P}_2, \mathsf{P}_1, y, x, f, H, G$

B.5.3.3 Characters

(I introduced these now as they fit nicely in defining subfield homomorphisms and will be needed in the development of the Theorem A proof.)

▷ Introduce S; $\oplus$: $\mathsf{map}_2\ S\ S\ S$; A : subset S and F : field

▷ Define is_not_all_zero = $[\lambda f : \mathsf{map}\ S\ F]\ \neg(\{\forall x : A\}\ (f\ x) = \mathbf{0})$
: $(\mathsf{map}\ S\ F) \to \mathsf{prop}$

▷ Prove not_all_zero_forms_subset : subset_axioms is_not_all_zero
$= [\lambda f_1, f_2 \,|\, \mathsf{map}\ S\ F]\ [\lambda Q f : f_1 = f_2]\ [\lambda H : f_1.\mathsf{is\_not\_all\_zero}]\ [\lambda X : \{\forall x : A\}$
$(f_2\ x) = \mathbf{0}]\ H\ ([\lambda x : A]\ (Q f\ x).\mathsf{tran}\ (X\ x))$

▷ Define not_all_zero

$= (\mathsf{is\_not\_all\_zero}, \mathsf{not\_all\_zero\_forms\_subset} : \mathsf{subset}\ (\mathsf{map}\ S\ F))$

$: \mathsf{subset}\ (\mathsf{map}\ S\ F)$

▷ Allow -character to be written postfix

▷ Define -character $= \mathsf{not\_all\_zero} \cap (\mathsf{resp\_map}_2 \oplus (\times|F)\ A) : \mathsf{subset}\ (\mathsf{map}\ S\ F)$

▷ Discharge $F, A, \oplus, S$

B.5.3.4 Morphisms between subfields

▷ Define subfield_hom $= [\lambda F]\ [\lambda L : \mathsf{subset}\ F]\ (\mathsf{subgroup\_hom}\ L) \cap$

$(\times\text{-character}\ L\ F) : \{\Pi F\}\ (\mathsf{subset}\ F) \to \mathsf{subset}\ (\mathsf{map}\ F\ F)$

▷ Introduce F and L | subfield F

▷ Prove subfield_hom_forms_subgroup_hom
$: \{\forall f : \mathsf{subfield\_hom}\ L\}\ f \in (\mathsf{subgroup\_hom}\ L)$
$= [\lambda f : \mathsf{subfield\_hom}\ L]\ f.\mathsf{ev.fst}$

▷ Define coercion sfhom_sghom

$= [\lambda f : \mathsf{subfield\_hom}\ L]\ \mathsf{make}\ (\mathsf{subfield\_hom\_forms\_subgroup\_hom}\ f)$

$: (\mathsf{subfield\_hom}\ L) \to \mathsf{subgroup\_hom}\ L$

▷ Introduce f : subfield_hom L and $x, y : L$

▷ Prove on_times : $(f\ (x \times y)) = ((f\ x) \times (f\ y))$
$= f.\mathsf{ev.snd.snd}\ x\ y$

▷ Discharge but keep y, x

▷ Suppose $H_1 : (f\ \mathbf{1}) = \mathbf{0}$

▷ Prove subresult $\mathsf{Q}_1 : (f\ x) = (f\ (x \times \mathbf{1}))$
$= f.\mathsf{resp}\ x.\mathsf{times\_un.symm}$

▷ Prove subresult $\mathsf{Q}_2 : (f\ (x \times (\mathbf{1}|L))) = ((f\ x) \times (f\ (\mathbf{1}|L)))$
$= \mathsf{on\_times}\ x\ (\mathbf{1}|L)$

▷ Prove subresult $\mathsf{Q}_3 : ((f\ x) \times (f\ \mathbf{1})) = ((f\ x) \times \mathbf{0})$
$= (f\ x).\mathsf{times}_2\ H_1$

▷ Prove subresult $\mathsf{Q}_4 : ((f\ x) \times \mathbf{0}) = \mathbf{0}$
$= (f\ x).\mathsf{times\_ze}$

▷ Prove subresult $\mathsf{Q}_5 : (f\ x) = \mathbf{0}$
$= (\mathsf{Q}_1.\mathsf{tran}\ \mathsf{Q}_2).\mathsf{tran}\ (\mathsf{Q}_3.\mathsf{tran}\ \mathsf{Q}_4)$

▷ Discharge H_1, y, x

▷ Prove subresult $\mathsf{C}_1 : (((f\ \mathbf{1}) = \mathbf{0}) \wedge ((^1\!/(f\ \mathbf{1})) = \mathbf{0})) \to (f\ \mathbf{1}) = \mathbf{1}$
$= [\lambda H : ((f\ \mathbf{1}) = \mathbf{0}) \wedge ((^1\!/(f\ \mathbf{1})) = \mathbf{0})]$
$\mathsf{ex\_falso}\ ((f\ \mathbf{1}) = \mathbf{1})\ (f.\mathsf{ev.snd.fst}\ ([\lambda x : L]\ \mathsf{Q}_5\ x\ H.\mathsf{fst}))$

▷ Suppose $H_2 : ((f\ \mathbf{1}) \times (^1\!/(f\ \mathbf{1}))) = \mathbf{1}$

▷ Prove subresult $\mathsf{Q}_6 : (f\ \mathbf{1}) = ((f\ \mathbf{1}) \times \mathbf{1})$
$= (f\ \mathbf{1}).\mathsf{times\_un.symm}$

▷ Prove subresult $\mathsf{Q}_7 : ((f\ \mathbf{1}) \times \mathbf{1}) = ((f\ \mathbf{1}) \times ((f\ \mathbf{1}) \times (^1\!/(f\ \mathbf{1}))))$
$= (f\ \mathbf{1}).\mathsf{times}_2\ H_2.\mathsf{symm}$

▷ Prove subresult Q_8
$: ((f\ \mathbf{1}) \times ((f\ \mathbf{1}) \times (^1\!/(f\ \mathbf{1})))) = (((f\ \mathbf{1}) \times (f\ \mathbf{1})) \times (^1\!/(f\ \mathbf{1})))$
$= (\mathsf{times\_assoc}\ (f\ \mathbf{1})\ (f\ \mathbf{1})\ (^1\!/(f\ \mathbf{1}))).\mathsf{symm}$

▷ Prove subresult Q_9
$: (((f\ (\mathbf{1}|L)) \times (f\ (\mathbf{1}|L))) \times (^1\!/(f\ \mathbf{1}))) = ((f\ ((\mathbf{1}|L) \times (\mathbf{1}|L))) \times (^1\!/(f\ \mathbf{1})))$
$= (\mathsf{on\_times}\ (\mathbf{1}|L)\ (\mathbf{1}|L)).\mathsf{symm.times}_1\ (^1\!/(f\ \mathbf{1}))$

▷ Prove subresult $\mathsf{Q}_{10} : ((f\ (\mathbf{1} \times \mathbf{1})) \times (^1\!/(f\ \mathbf{1}))) = ((f\ \mathbf{1}) \times (^1\!/(f\ \mathbf{1})))$
$= (f.\mathsf{resp}\ \mathbf{1}.\mathsf{times\_un}).\mathsf{times}_1\ (^1\!/(f\ \mathbf{1}))$

▷ Prove subresult $\mathsf{C}_2 : (f\ \mathbf{1}) = \mathbf{1}$
$= ((\mathsf{Q}_6.\mathsf{tran}\ \mathsf{Q}_7).\mathsf{tran}\ \mathsf{Q}_8).\mathsf{tran}\ (\mathsf{Q}_9.\mathsf{tran}\ (\mathsf{Q}_{10}.\mathsf{tran}\ H_2))$

▷ Discharge H_2

▷ Prove $\mathsf{on\_un} : (f\ \mathbf{1}) = \mathbf{1}$
$= \mathsf{case}\ (\mathsf{times\_recip}\ (f\ \mathbf{1}))\ \mathsf{C}_1\ \mathsf{C}_2$

▷ Discharge $\mathsf{C}_2, \mathsf{Q}_{10}, \mathsf{Q}_9, \mathsf{Q}_8, \mathsf{Q}_7, \mathsf{Q}_6, \mathsf{C}_1, \mathsf{Q}_5, \mathsf{Q}_4, \mathsf{Q}_3, \mathsf{Q}_2, \mathsf{Q}_1$

▷ Introduce $x : L$ and suppose $H_1 : x = \mathbf{0}$

▷ Prove subresult $\mathsf{Q}_1 : (f\ (^1\!/x)) = (f\ \mathbf{0})$
$= f.\mathsf{resp}\ ((^1\!/.\mathsf{resp}\ H_1).\mathsf{tran}\ \mathsf{ze\_recip})$

▷ Prove subresult $\mathsf{Q}_2 : \mathbf{0} = (^1\!/(f\ \mathbf{0}))$
$= ((^1\!/.\mathsf{resp}\ f.\mathsf{on\_ze}).\mathsf{tran}\ \mathsf{ze\_recip}).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_3 : (^1\!/(f\ \mathbf{0})) = (^1\!/(f\ x))$
$= {}^1\!/.\mathsf{resp}\ (f.\mathsf{resp}\ H_1.\mathsf{symm})$

▷ Prove subresult $\mathsf{P}_1 : (f\ ({}^1\!/x)) = ({}^1\!/(f\ x))$
$= (\mathsf{Q}_1.\mathsf{tran}\ f.\mathsf{on\_ze}).\mathsf{tran}\ (\mathsf{Q}_2.\mathsf{tran}\ \mathsf{Q}_3)$

▷ Discharge H_1

▷ Prove subresult $\mathsf{C}_1 : ((x = \mathbf{0}) \wedge (({}^1\!/x) = \mathbf{0})) \to (f\ ({}^1\!/x)) = ({}^1\!/(f\ x))$
$= [\lambda H : (x = \mathbf{0}) \wedge (({}^1\!/x) = \mathbf{0})]\ \mathsf{P}_1\ H.\mathsf{fst}$

▷ Suppose $H_2 : (x \times ({}^1\!/x)) = \mathbf{1}$

▷ Suppose $H_3 : ((f\ x) = \mathbf{0}) \wedge (({}^1\!/(f\ x)) = \mathbf{0})$

▷ Prove subresult $\mathsf{Q}_4 : \mathbf{1} = (f\ \mathbf{1})$
$= \mathsf{on\_un.symm}$

▷ Prove subresult $\mathsf{Q}_5 : (f\ \mathbf{1}) = (f\ (x \times ({}^1\!/x)))$
$= f.\mathsf{resp}\ H_2.\mathsf{symm}$

▷ Prove subresult Q_6
$: (f\ (x \times (\mathsf{recip\_closed}\ x))) = ((f\ x) \times (f\ (\mathsf{recip\_closed}\ x)))$
$= \mathsf{on\_times}\ x\ (\mathsf{recip\_closed}\ x)$

▷ Prove subresult $\mathsf{Q}_7 : ((f\ x) \times (f\ ({}^1\!/x))) = (\mathbf{0} \times (f\ ({}^1\!/x)))$
$= H_3.\mathsf{fst}.\mathsf{times}_1\ (f\ ({}^1\!/x))$

▷ Prove subresult $\mathsf{Q}_8 : (\mathbf{0} \times (f\ ({}^1\!/x))) = \mathbf{0}$
$= \mathsf{ze\_times}\ (f\ ({}^1\!/x))$

▷ Prove subresult $\mathsf{C}_{2\text{-}1} : (f\ ({}^1\!/x)) = ({}^1\!/(f\ x))$
$= \mathsf{ex\_falso}\ ((f\ ({}^1\!/x)) = ({}^1\!/(f\ x)))$
$(\mathsf{nontriv}\ ((\mathsf{Q}_4.\mathsf{tran}\ \mathsf{Q}_5).\mathsf{tran}\ (\mathsf{Q}_6.\mathsf{tran}\ (\mathsf{Q}_7.\mathsf{tran}\ \mathsf{Q}_8))))$

▷ Discharge H_3

▷ Suppose $H_4 : ((f\ x) \times ({}^1\!/(f\ x))) = \mathbf{1}$

▷ Prove subresult $\mathsf{Q}_9 : ({}^1\!/(f\ x)) = (({}^1\!/(f\ x)) \times \mathbf{1})$
$= ({}^1\!/(f\ x)).\mathsf{times\_un.symm}$

▷ Prove subresult $\mathsf{Q}_{10} : \mathbf{1} = (f\ (x \times ({}^1\!/x)))$
$= ((f.\mathsf{resp}\ H_2).\mathsf{tran}\ \mathsf{on\_un}).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_{11} : (f\ (x \times ({}^1\!/x))) = ((f\ x) \times (f\ ({}^1\!/x)))$
$= \mathsf{on\_times}\ x\ (\mathsf{recip\_closed}\ x)$

▷ Prove subresult $\mathsf{Q}_{12} : (({}^1\!/(f\ x)) \times \mathbf{1}) = (({}^1\!/(f\ x)) \times ((f\ x) \times (f\ ({}^1\!/x))))$
$= ({}^1\!/(f\ x)).\mathsf{times}_2\ (\mathsf{Q}_{10}.\mathsf{tran}\ \mathsf{Q}_{11})$

▷ Prove subresult Q_{13}
$: (({}^1\!/(f\ x)) \times ((f\ x) \times (f\ ({}^1\!/x)))) = ((({}^1\!/(f\ x)) \times (f\ x)) \times (f\ ({}^1\!/x)))$
$= (\mathsf{times\_assoc}\ ({}^1\!/(f\ x))\ (f\ x)\ (f\ ({}^1\!/x))).\mathsf{symm}$

▷ Prove subresult Q_{14} : $((^1/(f\ x)) \times (f\ x)) = ((f\ x) \times (^1/(f\ x)))$
= times_comm $(^1/(f\ x))$ $(f\ x)$

▷ Prove subresult Q_{15} : $(((^1/(f\ x)) \times (f\ x)) \times (f\ (^1/x))) = (\mathbf{1} \times (f\ (^1/x)))$
= $(\mathsf{Q}_{14}$.tran $H_4)$.times$_1$ $(f\ (^1/x))$

▷ Prove subresult Q_{16} : $(\mathbf{1} \times (f\ (^1/x))) = (f\ (^1/x))$
= un_times $(f\ (^1/x))$

▷ Prove subresult $\mathsf{C}_{2\text{-}2}$: $(f\ (^1/x)) = (^1/(f\ x))$
= $((\mathsf{Q}_9$.tran $\mathsf{Q}_{12})$.tran $(\mathsf{Q}_{13}$.tran $(\mathsf{Q}_{15}$.tran $\mathsf{Q}_{16})))$.symm

▷ Discharge H_4

▷ Prove subresult C_2 : $(f\ (^1/x)) = (^1/(f\ x))$
= case (times_recip $(f\ x)$) $\mathsf{C}_{2\text{-}1}$ $\mathsf{C}_{2\text{-}2}$

▷ Discharge H_2

▷ Prove on_recip : $(f\ (^1/x)) = (^1/(f\ x))$
= case (times_recip x) C_1 C_2

▷ Discharge $\mathsf{C}_2, \mathsf{C}_{2\text{-}2}, \mathsf{Q}_{16}, \mathsf{Q}_{15}, \mathsf{Q}_{14}, \mathsf{Q}_{13}, \mathsf{Q}_{12}, \mathsf{Q}_{11}, \mathsf{Q}_{10}, \mathsf{Q}_9, \mathsf{C}_{2\text{-}1}, \mathsf{Q}_8, \mathsf{Q}_7, \mathsf{Q}_6, \mathsf{Q}_5,$
$\mathsf{Q}_4, \mathsf{C}_1, \mathsf{P}_1, \mathsf{Q}_3, \mathsf{Q}_2, \mathsf{Q}_1, x, f, L, F$

B.5.4 Decideable substructures

B.5.4.1 Decideable subgroups

▷ Define dsubgroup = $[\lambda G]\ \langle \Sigma J : \text{subgroup}\ G \rangle\ J$.is_decideable : group $\to$ Type$_1$

▷ Define coercion undecide_subgroup = $[\lambda G \,|\, \text{group}]\ [\lambda J : \text{dsubgroup}\ G]\ J_{.\mathbf{1}}$
: $\{\Pi G \,|\, \text{group}\}$ (dsubgroup G) $\to$ subgroup G

▷ Define coercion dsubcar = $[\lambda G \,|\, \text{group}]\ [\lambda J : \text{dsubgroup}\ G]$
$(J, J_{.\mathbf{2}} : \text{dsubset}\ G)$: $\{\Pi G \,|\, \text{group}\}$ (dsubgroup G) $\to$ dsubset G

B.5.4.2 Decideable subfields

▷ Define dsubfield = $[\lambda F : \text{field}]\ \langle \Sigma K \rangle\ K$.is_decideable : field $\to$ Type$_1$

▷ Define coercion undecide_subfield = $[\lambda F]\ [\lambda K : \text{dsubfield}\ F]\ K_{.\mathbf{1}}$
: $\{\Pi F\}$ (dsubfield F) $\to$ subfield F

▷ Define coercion dsubfg $= [\lambda F]\ [\lambda K : \mathsf{dsubfield}\ F]\ (K, K_{.\mathbf{2}} : \mathsf{dsubgroup}\ F)$
$: \{\Pi F\}\ (\mathsf{dsubfield}\ F) \to \mathsf{dsubgroup}\ F$

▷ Introduce F and L

▷ Define dsubfield_of $= \langle \Sigma K : \mathsf{dsubfield}\ F \rangle\ K \subseteq L : \mathsf{Type}_1$

▷ Define coercion undsubf $= [\lambda K : \mathsf{dsubfield\_of}]\ K_{.\mathbf{1}} : \mathsf{dsubfield\_of} \to \mathsf{dsubfield}\ F$

▷ Define coercion dsub_subf_of $= [\lambda K : \mathsf{dsubfield\_of}]\ (K_{.\mathbf{1}.\mathbf{1}}, K_{.\mathbf{2}} : \mathsf{subfield\_of}\ L)$
$: \mathsf{dsubfield\_of} \to \mathsf{subfield\_of}\ L$

▷ Define coercion make_dsubf $= [\lambda K \mid \mathsf{dsubfield}\ F]\ [\lambda H : K \subseteq L]$
$(K, H : \mathsf{dsubfield\_of}) : \{\Pi K \mid \mathsf{dsubfield}\ F\}\ (K \subseteq L) \to \mathsf{dsubfield\_of}$

▷ Discharge L, F

B.5.5 Quotients involving groups

B.5.5.1 Conjugation within a group

▷ Introduce $G \mid \mathsf{group}$

▷ Construct by refinement conj_forms_map$_2$ $: ([\lambda x, y : G]\ y^{-1} \circ (x \circ y))$.is_map$_2$
$\scriptstyle (\mathsf{o\_resp},\ \circ,\ \mathsf{inv\_resp},\ ^{-1},\ =)$

▷ Allow $\divideontimes$ to be written infix

▷ Define $\divideontimes = ([\lambda x, y : G]\ y^{-1} \circ (x \circ y), \mathsf{conj\_forms\_map}_2 : \mathsf{map}_2\ G\ G\ G)$
$: \mathsf{map}_2\ G\ G\ G$

▷ Discharge G

B.5.5.2 Definition of normality of subgroups

▷ Define normal_in $= [\lambda G \mid \mathsf{group}]\ [\lambda H, J : \mathsf{subgroup}\ G]\ \{\forall j : J\}\ \{\forall h : H\}$
$(h \divideontimes j) \in H : \{\Pi G \mid \mathsf{group}\}\ (\mathsf{subgroup}\ G) \to (\mathsf{subgroup}\ G) \to \mathsf{prop}$

B.5.5.3 Cosets of a subgroup

▷ Introduce $G \mid \mathsf{group}$ and U

▷ Construct by refinement same_right_coset_of_forms_equiv_rel
: is_equiv_rel $([\lambda x, y : G]\ (x^{-1} \circ y) \in U)$
(o_closed, id_o, o_inv, id, $^{-1}$, $\circ$, o$_1$, tran, o_assoc, symm, o$_2$, eq_closed, $\in$,
inv_closed, inv_inv, inv_of_o, id_closed, inv_o, $=$, is_tran, is_symm, pair$_3$)

▷ Allow $\approx$- to be written prefix

▷ Define $\approx$- = $([\lambda x, y : G]\ (x^{-1} \circ y) \in U$, same_right_coset_of_forms_equiv_rel :
equiv_rel G) : equiv_rel G

▷ Discharge U, G

▷ Define coercion coset_co = $\approx$- : $\{\Pi G \mid \text{group}\}$ (subgroup G) $\to$ equiv_rel G

B.5.5.4 Results concerning taking cosets of a subgroup

▷ Introduce G | group and D, U | subgroup G

▷ Prove COSET$_1$s : $(D \subseteq U) \to (\approx\text{-}D)$.is_subrelation $(\approx\text{-}U)$
$= [\lambda H : D \subseteq U]\ [\lambda x, y \mid G]\ [\lambda Q : x.(\approx\text{-}D)\ y]\ H$ (make Q)

▷ Discharge but keep U, D

▷ Prove COSET$_1$: $(D = U) \to (\approx\text{-}D)$.is_same_relation $(\approx\text{-}U)$
$= [\lambda Q : D = U]$ pair (COSET$_1$s Q.fst) (COSET$_1$s Q.snd)

▷ Discharge U, D, G

B.5.5.5 Results concerning quotients and cosets

▷ Introduce S; A : subset S and $\sim, \sim'$ | equiv_rel S

▷ Suppose H : $\sim$.is_subrelation $\sim'$

▷ Define quot_fun = $[\lambda x : A/\sim]$ $(x.\text{rep}, (x.\text{ev})_{.\mathbf{1}}, H\ (x.\text{ev})_{.\mathbf{2}} : A/\sim')$
: $(A/\sim) \to A/\sim'$

▷ Prove quot_forms_map : $\{\forall x, y \mid A/\sim\}\ (x = y) \to (\text{quot\_fun}\ x) = (\text{quot\_fun}\ y)$
$= [\lambda x, y \mid A/\sim]\ [\lambda Q : x = y]\ H|x.\text{rep}|y.\text{rep}\ Q$

▷ Define quot_map = (quot_fun, quot_forms_map : map $(A/\sim)$ $(A/\sim')$)
: map $(A/\sim)$ $(A/\sim')$

▷ Discharge $H, \sim', \sim$

▷ Prove subresult P_1 : $\{\forall \sim, \sim' \mid \mathsf{equiv\_rel}\ S\}\ \{\Pi Q : \sim.\mathsf{is\_same\_relation}\ \sim'\}$
$((\mathsf{quot\_map}\ Q.\mathsf{snd}) \circ (\mathsf{quot\_map}\ Q.\mathsf{fst})) = \mathsf{identity}$
$= [\lambda \sim, \sim' \mid \mathsf{equiv\_rel}\ S]\ [\lambda \_ : \sim.\mathsf{is\_same\_relation}\ \sim']\ \mathsf{refl}|(A/\sim)$

▷ Introduce $\sim, \sim' \mid \mathsf{equiv\_rel}\ S$ and Q : $\sim.\mathsf{is\_same\_relation}\ \sim'$

▷ Prove $\mathsf{quot\_forms\_iso}$: $(\mathsf{quot\_map}\ Q.\mathsf{fst}) \in (\mathsf{iso}\ (A/\sim)\ (A/\sim'))$
$= (\mathsf{quot\_map}\ Q.\mathsf{snd}, \mathsf{pair}\ (\mathsf{P}_1\ Q)\ (\mathsf{P}_1\ (\mathsf{pair}\ Q.\mathsf{snd}\ Q.\mathsf{fst})) :$
$(\mathsf{quot\_map}\ Q.\mathsf{fst}) \in (\mathsf{iso}\ (A/\sim)\ (A/\sim')))$

▷ Define $\mathsf{quot\_iso} = (\mathsf{quot\_map}\ Q.\mathsf{fst}, \mathsf{quot\_forms\_iso} : \mathsf{iso}\ (A/\sim)\ (A/\sim'))$
: $\mathsf{iso}\ (A/\sim)\ (A/\sim')$

▷ Discharge $Q, \sim', \sim$

▷ Construct by refinement $\mathsf{quot\_triv\_iso}$: $\mathsf{iso}\ (A/=)\ A$
$(=, \mathsf{refl}, \mathsf{rep}, =, /, /, \in, \mathsf{is\_map}, \mathsf{ev}, \mathsf{symm}, \mathsf{tran}, \mathsf{subset\_forms\_set}, \mathsf{set}, \circ,$
$\mathsf{identity}, \mathsf{map}, \mathsf{pair}, \mathsf{iso})$

▷ Discharge P_1, A, S

B.6 Further work on set mappings

B.6.1 Various tools for use with mappings

B.6.1.1 Currying of mappings

▷ Introduce $S, T, U \mid \mathsf{set}$

▷ Let $\mathsf{curry\_fun} = [\lambda f : \mathsf{map}_2\ S\ T\ U]\ [\lambda x : S]\ (f\ x, f.\mathsf{resps}_2\ x : \mathsf{map}\ T\ U)$
: $(\mathsf{map}_2\ S\ T\ U) \to S \to \mathsf{map}\ T\ U$

▷ Let $\mathsf{curry\_map}$
$= [\lambda f : \mathsf{map}_2\ S\ T\ U]\ (\mathsf{curry\_fun}\ f, f.\mathsf{resps}_1 : \mathsf{map}\ S\ (\mathsf{map}\ T\ U))$
: $(\mathsf{map}_2\ S\ T\ U) \to \mathsf{map}\ S\ (\mathsf{map}\ T\ U)$

▷ Prove $\mathsf{curry\_forms\_map}$: $\mathsf{curry\_map.is\_map}$
$= [\lambda f_1, f_2 \mid \mathsf{map}_2\ S\ T\ U]\ [\lambda Q : f_1 = f_2]\ Q$

▷ Define curry

$= (\mathsf{curry\_map}, \mathsf{curry\_forms\_map} : \mathsf{map}\ (\mathsf{map}_2\ S\ T\ U)\ (\mathsf{map}\ S\ (\mathsf{map}\ T\ U)))$

$: \mathsf{map}\ (\mathsf{map}_2\ S\ T\ U)\ (\mathsf{map}\ S\ (\mathsf{map}\ T\ U))$

▷ Discharge $\mathsf{curry\_map}, \mathsf{curry\_fun}, U, T, S$

B.6.1.2 The subset fixed by a mapping

▷ Introduce S and $A : \mathsf{subset}\ S$

▷ Define $\mathsf{fixes} = [\lambda f : \mathsf{map}\ S\ S]\ [\lambda x : S]\ (f\ x) = x : (\mathsf{map}\ S\ S) \to S \to \mathsf{prop}$

▷ Prove $\mathsf{fixing\_forms\_subset} : \mathsf{subset\_axioms}\ ([\lambda f : \mathsf{map}\ S\ S]\ \{\forall x : A\}\ f.\mathsf{fixes}\ x)$
$= [\lambda f_1, f_2 \mid \mathsf{map}\ S\ S]\ [\lambda Q : f_1 = f_2]\ [\lambda H : \{\forall x : A\}\ f_1.\mathsf{fixes}\ x]\ [\lambda x : A]$
$(Q\ x).\mathsf{symm.tran}\ (H\ x)$

▷ Define fixing

$= ([\lambda f : \mathsf{map}\ S\ S]\ \{\forall x : A\}\ f.\mathsf{fixes}\ x, \mathsf{fixing\_forms\_subset} : \mathsf{subset}\ (\mathsf{map}\ S\ S))$

$: \mathsf{subset}\ (\mathsf{map}\ S\ S)$

▷ Discharge A, S

B.6.1.3 Mappings from a particular subdomain of interest

▷ Introduce S and $A : \mathsf{subset}\ S$

▷ Define $\mathsf{is\_map\_from} = [\lambda f : \mathsf{map}\ S\ S]\ \{\Pi x : S\}\ (x \in A) \vee (f.\mathsf{fixes}\ x)$

$: (\mathsf{map}\ S\ S) \to \mathsf{prop}$

▷ Construct by refinement $\mathsf{map\_from\_forms\_subset} : \mathsf{subset\_axioms}\ \mathsf{is\_map\_from}$
(is_map, symm, tran, $=$, $\in$, inr, fixes, inl, $\vee$, case, is_map_from, map)

▷ Define $\mathsf{map\_from} = (\mathsf{is\_map\_from}, \mathsf{map\_from\_forms\_subset} : \mathsf{subset}\ (\mathsf{map}\ S\ S))$

$: \mathsf{subset}\ (\mathsf{map}\ S\ S)$

▷ Discharge A, S

B.6.1.4 Results concerning mappings from particular subdomains

▷ Introduce S and A, B

▷ Prove $\mathsf{FROM_1s} : (A \subseteq B) \to (\mathsf{map\_from}\ A) \subseteq (\mathsf{map\_from}\ B)$
$= [\lambda H : A \subseteq B]\ [\lambda f : \mathsf{map\_from}\ A]\ [\lambda x : S]$
$\mathsf{case}\ (f.\mathsf{ev}\ x)\ ([\lambda H_1 : x \in A]\ \mathsf{inl}|(x \in B)|((f\ x) = x)\ (H\ H_1))\ \mathsf{inr}$

▷ Discharge but keep B, A

▷ Prove $\mathsf{FROM_1} : (A = B) \to (\mathsf{map\_from}\ A) = (\mathsf{map\_from}\ B)$
$= [\lambda Q : A = B]\ \mathsf{pair}\ (\mathsf{FROM_1s}\ Q.\mathsf{fst})\ (\mathsf{FROM_1s}\ Q.\mathsf{snd})$

▷ Discharge B, A, S

B.6.1.5 Mappings that agree on a part of their domain

▷ Introduce S, T; $A : \mathsf{subset}\ S$ and $f_1, f_2 : \mathsf{map}\ S\ T$

▷ Prove $\mathsf{agree\_forms\_subset}$
$: \{\forall x_1, x_2 \,|\, S\}\ (x_1 = x_2) \to ((f_1\ x_1) = (f_2\ x_1)) \to (f_1\ x_2) = (f_2\ x_2)$
$= [\lambda x_1, x_2 \,|\, S]\ [\lambda Q : x_1 = x_2]\ [\lambda X_1 : (f_1\ x_1) = (f_2\ x_1)]$
$(f_1.\mathsf{resp}\ Q.\mathsf{symm}).\mathsf{tran}\ (X_1.\mathsf{tran}\ (f_2.\mathsf{resp}\ Q))$

▷ Define $\mathsf{agree} = ([\lambda x : S]\ (f_1\ x) = (f_2\ x), \mathsf{agree\_forms\_subset} : \mathsf{subset}\ S)$
$: \mathsf{subset}\ S$

▷ Let $\sim\ = A \subseteq \mathsf{agree} : \mathsf{prop}$

▷ Discharge f_2, f_1

▷ Prove subresult $\mathsf{q\_agrees} : \{\forall f_1, f_2 \,|\, \mathsf{map}\ S\ T\}\ (f_1 = f_2) \to f_1 \sim f_2$
$= [\lambda f_1, f_2 \,|\, \mathsf{map}\ S\ T]\ [\lambda Q : f_1 = f_2]\ [\lambda x : A]\ Q\ x$

▷ Prove subresult $\mathsf{q\_symm} : \{\forall f_1, f_2 \,|\, \mathsf{map}\ S\ T\}\ (f_1 \sim f_2) \to f_2 \sim f_1$
$= [\lambda f_1, f_2 \,|\, \mathsf{map}\ S\ T]\ [\lambda Q : f_1 \sim f_2]\ [\lambda x : A]\ (Q\ x).\mathsf{symm}$

▷ Prove subresult $\mathsf{q\_tran}$
$: \{\forall f_1, f_2, f_3 \,|\, \mathsf{map}\ S\ T\}\ (f_1 \sim f_2) \to (f_2 \sim f_3) \to f_1 \sim f_3$
$= [\lambda f_1, f_2, f_3 \,|\, \mathsf{map}\ S\ T]\ [\lambda Q_1 : f_1 \sim f_2]\ [\lambda Q_2 : f_2 \sim f_3]\ [\lambda x : A]$
$(Q_1\ x).\mathsf{tran}\ (Q_2\ x)$

▷ Prove $\mathsf{eq\_maps\_wrt\_forms\_equiv\_rel} : \sim.\mathsf{is\_equiv\_rel}$
$= \mathsf{pair_3}\ \mathsf{q\_agrees}\ \mathsf{q\_symm}\ \mathsf{q\_tran}$

▷ Define $\mathsf{eq\_maps\_wrt} = ([\lambda f_1, f_2 : \mathsf{map}\ S\ T]\ A \subseteq (f_1.\mathsf{agree}\ f_2),$
$\mathsf{eq\_maps\_wrt\_forms\_equiv\_rel} : \mathsf{equiv\_rel}\ (\mathsf{map}\ S\ T)) : \mathsf{equiv\_rel}\ (\mathsf{map}\ S\ T)$

▷ Define $\mathsf{map\_wrt} = (\mathsf{map}\ S\ T)/\mathsf{eq\_maps\_wrt} : \mathsf{set}$

▷ Allow $=\!|$ to be written midfix

▷ Define $=\!| = \mathsf{equal\_in}\ \mathsf{map\_wrt} : \mathsf{rel}\ (\mathsf{map}\ S\ T)$

▷ Discharge $\mathsf{q\_tran}, \mathsf{q\_symm}, \mathsf{q\_agrees}, \sim, A, T, S$

B.6.1.6 Applying a mapping across a subset

▷ Introduce S, T; f **:** map S T and A **:** subset S

▷ Define $\mathsf{is\_apply\_across} = [\lambda x : T]\ \langle \exists x' : A \rangle\ x = (f\ x') : T \to \mathsf{prop}$

▷ Construct by refinement $\mathsf{apply\_across\_forms\_subset}$
: subset_axioms is_apply_across
(symm, tran, =, is_apply_across)

▷ Define $\mathsf{apply\_across} = (\mathsf{is\_apply\_across}, \mathsf{apply\_across\_forms\_subset} : \mathsf{subset}\ T)$
: subset T

▷ Discharge A, f, T, S

B.6.1.7 Results concerning applying mappings across subsets

▷ Introduce $S, T, U \mid \mathsf{set}$ and f **:** map S T

▷ Prove AA_0 **:** $\{\forall A\}\ \{\forall x : A\}\ (f\ x) \in (f.\mathsf{apply\_across}\ A)$
$= [\lambda A]\ [\lambda x : A]\ (x, \mathsf{refl}\ (f\ x) : (f\ x) \in (f.\mathsf{apply\_across}\ A))$

▷ Prove $\mathsf{AA}_2\mathsf{s}$ **:** $\{\forall A, B\}\ (A \subseteq B) \to (f.\mathsf{apply\_across}\ A) \subseteq (f.\mathsf{apply\_across}\ B)$
$= [\lambda A, B]\ [\lambda H : A \subseteq B]\ [\lambda x : f.\mathsf{apply\_across}\ A]$
$((x.\mathsf{ev})._{\mathbf{1}}.\mathsf{rep}, H\ (x.\mathsf{ev})._{\mathbf{1}}.\mathsf{ev} : B), (x.\mathsf{ev})._{\mathbf{2}} : x \in (f.\mathsf{apply\_across}\ B))$

▷ Prove AA_2 **:** $\{\forall A, B\}\ (A = B) \to (f.\mathsf{apply\_across}\ A) = (f.\mathsf{apply\_across}\ B)$
$= [\lambda A, B]\ [\lambda Q : A = B]\ \mathsf{pair}\ (\mathsf{AA}_2\mathsf{s}\ Q.\mathsf{fst})\ (\mathsf{AA}_2\mathsf{s}\ Q.\mathsf{snd})$

▷ Suppose H **:** $\{\forall x_1, x_2 \mid S\}\ ((f\ x_1) = (f\ x_2)) \to x_1 = x_2$

▷ Prove $\mathsf{AA}_2\mathsf{s'}$ **:** $\{\forall A, B\}\ ((f.\mathsf{apply\_across}\ A) \subseteq (f.\mathsf{apply\_across}\ B)) \to A \subseteq B$
$= [\lambda A, B]\ [\lambda H_1 : (f.\mathsf{apply\_across}\ A) \subseteq (f.\mathsf{apply\_across}\ B)]\ [\lambda x : A]$
$[\delta \mathsf{fx} = (f\ x, \mathsf{AA}_0\ x : f.\mathsf{apply\_across}\ A)]$
$\mathsf{eq\_closed}\ (H\ ((H_1\ \mathsf{fx}).\mathsf{ev})._{\mathbf{2}}).\mathsf{symm}\ ((H_1\ \mathsf{fx}).\mathsf{ev})._{\mathbf{1}}.\mathsf{ev}$

▷ Prove AA_2' **:** $\{\forall A, B\}\ ((f.\mathsf{apply\_across}\ A) = (f.\mathsf{apply\_across}\ B)) \to A = B$
$= [\lambda A, B]\ [\lambda Q : (f.\mathsf{apply\_across}\ A) = (f.\mathsf{apply\_across}\ B)]$
$\mathsf{pair}\ (\mathsf{AA}_2\mathsf{s'}\ Q.\mathsf{fst})\ (\mathsf{AA}_2\mathsf{s'}\ Q.\mathsf{snd})$

▷ Discharge H, f

▷ Prove $\mathsf{AA}_1\mathsf{s}$ **:** $\{\forall f_1, f_2 \mid \mathsf{map}\ S\ T\}\ \{\forall A\}\ (f_1.(=\restriction A)\ f_2) \to$
$(f_1.\mathsf{apply\_across}\ A) \subseteq (f_2.\mathsf{apply\_across}\ A)$
$= [\lambda f_1, f_2 \mid \mathsf{map}\ S\ T]\ [\lambda A]\ [\lambda Q : f_1.(=\restriction A)\ f_2]\ [\lambda x : f_1.\mathsf{apply\_across}\ A]$
$((x.\mathsf{ev})._{\mathbf{1}}, (x.\mathsf{ev})._{\mathbf{2}}.\mathsf{tran}\ (Q\ (x.\mathsf{ev})._{\mathbf{1}}) : x \in (f_2.\mathsf{apply\_across}\ A))$

▷ Prove AA_1 : $\{\forall f_1, f_2 \,|\, \mathsf{map}\ S\ T\}\ \{\forall A\}\ (f_1.(=\restriction A)\ f_2) \to$
$(f_1.\mathsf{apply\_across}\ A) = (f_2.\mathsf{apply\_across}\ A)$
$= [\lambda f_1, f_2 \,|\, \mathsf{map}\ S\ T]\ [\lambda A]\ [\lambda Q : f_1.(=\restriction A)\ f_2]\ \mathsf{pair}\ (\mathsf{AA}_1\mathsf{s}\ Q)\ (\mathsf{AA}_1\mathsf{s}\ Q.\mathsf{symm})$

▷ Introduce A : subset S; f : map $S\ T$ and g : map $T\ U$

▷ Prove AA_3 : $((g \circ f).\mathsf{apply\_across}\ A) = (g.\mathsf{apply\_across}\ (f.\mathsf{apply\_across}\ A))$
$= \mathsf{pair}\ ([\lambda x : (g \circ f).\mathsf{apply\_across}\ A]$
$((f\ (x.\mathsf{ev}).\mathbf{1}, \mathsf{AA}_0\ f\ (x.\mathsf{ev}).\mathbf{1} : f.\mathsf{apply\_across}\ A), (x.\mathsf{ev}).\mathbf{2} :$
$x \in (g.\mathsf{apply\_across}\ (f.\mathsf{apply\_across}\ A))))$
$([\lambda x : g.\mathsf{apply\_across}\ (f.\mathsf{apply\_across}\ A)]\ (((x.\mathsf{ev}).\mathbf{1}.\mathsf{ev}).\mathbf{1},$
$(x.\mathsf{ev}).\mathbf{2}.\mathsf{tran}\ (g.\mathsf{resp}\ ((x.\mathsf{ev}).\mathbf{1}.\mathsf{ev}).\mathbf{2}) : x \in ((g \circ f).\mathsf{apply\_across}\ A)))$

▷ Introduce B : subset T

▷ Prove subresult $\mathsf{AA}_{4\text{-}1}$: $((f.\mathsf{apply\_across}\ A) \subseteq B) \to \{\forall x : A\}\ (f\ x) \in B$
$= [\lambda H : (f.\mathsf{apply\_across}\ A) \subseteq B]\ [\lambda x : A]\ H\ (f\ x, \mathsf{AA}_0\ f\ x : f.\mathsf{apply\_across}\ A)$

▷ Prove subresult $\mathsf{AA}_{4\text{-}2}$: $(\{\forall x : A\}\ (f\ x) \in B) \to (f.\mathsf{apply\_across}\ A) \subseteq B$
$= [\lambda H : \{\forall x : A\}\ (f\ x) \in B]\ [\lambda x : f.\mathsf{apply\_across}\ A]$
$\mathsf{eq\_closed}\ (x.\mathsf{ev}).\mathbf{2}.\mathsf{symm}\ (H\ (x.\mathsf{ev}).\mathbf{1})$

▷ Prove AA_4 : $((f.\mathsf{apply\_across}\ A) \subseteq B) \leftrightarrow (\{\forall x : A\}\ (f\ x) \in B)$
$= \mathsf{pair}\ \mathsf{AA}_{4\text{-}1}\ \mathsf{AA}_{4\text{-}2}$

▷ Suppose H : $\{\forall x_1, x_2 \,|\, A\}\ ((f\ x_1) = (f\ x_2)) \to x_1 = x_2$

▷ Let $\mathsf{AA}_5\mathsf{\_fun} = [\lambda x : A]\ (f\ x, x, \mathsf{refl}\ (f\ x) : f.\mathsf{apply\_across}\ A)$
: $A \to f.\mathsf{apply\_across}\ A$

▷ Prove subresult $\mathsf{AA}_5\mathsf{\_forms\_map}$: $\{\forall x_1, x_2 \,|\, A\}\ (x_1 = x_2) \to (f\ x_1) = (f\ x_2)$
$= [\lambda x_1, x_2 \,|\, A]\ [\lambda Q : x_1 = x_2]\ f.\mathsf{resp}\ Q$

▷ Let $\mathsf{AA}_5\mathsf{\_map} = (\mathsf{AA}_5\mathsf{\_fun}, \mathsf{AA}_5\mathsf{\_forms\_map} : \mathsf{map}\ A\ (f.\mathsf{apply\_across}\ A))$
: map A $(f.\mathsf{apply\_across}\ A)$

▷ Define $\mathsf{AA}_5\mathsf{\_inv\_fun} = [\lambda x : f.\mathsf{apply\_across}\ A]\ (x.\mathsf{ev}).\mathbf{1}$
: $(f.\mathsf{apply\_across}\ A) \to A$

▷ Prove $\mathsf{AA}_5\mathsf{\_inv\_forms\_map}$
: $\{\forall x_1, x_2 \,|\, f.\mathsf{apply\_across}\ A\}\ (x_1 = x_2) \to (x_1.\mathsf{ev}).\mathbf{1} = (x_2.\mathsf{ev}).\mathbf{1}$
$= [\lambda x_1, x_2 \,|\, f.\mathsf{apply\_across}\ A]\ [\lambda Q : x_1 = x_2]$
$H\ (((x_1.\mathsf{ev}).\mathbf{2}.\mathsf{symm.tran}\ Q).\mathsf{tran}\ (x_2.\mathsf{ev}).\mathbf{2})$

▷ Define $\mathsf{AA}_5\mathsf{\_inv\_map}$
$= (\mathsf{AA}_5\mathsf{\_inv\_fun}, \mathsf{AA}_5\mathsf{\_inv\_forms\_map} : \mathsf{map}\ (f.\mathsf{apply\_across}\ A)\ A)$
: map $(f.\mathsf{apply\_across}\ A)\ A$

▷ Prove subresult P_1 : $(\mathsf{AA}_5\_\mathsf{inv\_map} \circ \mathsf{AA}_5\_\mathsf{map}) = \mathsf{identity}$
= $[\lambda x : A]$ refl x

▷ Prove subresult P_2 : $(\mathsf{AA}_5\_\mathsf{map} \circ \mathsf{AA}_5\_\mathsf{inv\_map}) = \mathsf{identity}$
= $[\lambda x : f.\mathsf{apply\_across}\ A]\ (x.\mathsf{ev}).\mathbf{2}.\mathsf{symm}$

▷ Prove $\mathsf{AA}_5\_\mathsf{forms\_iso}$: $\mathsf{AA}_5\_\mathsf{inv\_map} \in (\mathsf{iso}\ (f.\mathsf{apply\_across}\ A)\ A)$
= $(\mathsf{AA}_5\_\mathsf{map}, \mathsf{pair}\ \mathsf{P}_2\ \mathsf{P}_1 : \mathsf{AA}_5\_\mathsf{inv\_map} \in (\mathsf{iso}\ (f.\mathsf{apply\_across}\ A)\ A))$

▷ Define apply_across_iso

= $(\mathsf{AA}_5\_\mathsf{inv\_map}, \mathsf{AA}_5\_\mathsf{forms\_iso} : \mathsf{iso}\ (f.\mathsf{apply\_across}\ A)\ A)$

: $\mathsf{iso}\ (f.\mathsf{apply\_across}\ A)\ A$

▷ Prove AA_5 : $\mathsf{iso}\ A\ (f.\mathsf{apply\_across}\ A)$
= apply_across_iso.iso_symm

▷ Discharge $\mathsf{P}_2, \mathsf{P}_1, \mathsf{AA}_5\_\mathsf{map}, \mathsf{AA}_5\_\mathsf{forms\_map}, \mathsf{AA}_5\_\mathsf{fun}, H, \mathsf{AA}_{4\_2}, \mathsf{AA}_{4\_1}, B, g, f, A, U, T, S$

B.6.2 Restriction of mappings

B.6.2.1 Definition of map restriction

▷ Introduce S; A : dsubset S; f : map $S\ S$ and x : S

▷ Construct by refinement restriction_makes_map
: $([\lambda x : S]\ \mathsf{if}\ (x \in_? A)\ (f\ x)\ x).\mathsf{is\_map}$
($\notin$, symm, eq_closed, $\in$, inl, if, $=$, ex_falso, $\in_?$, $\vee$, case_elim, inr, resp)

▷ Define restriction_fun

= $([\lambda x : S]\ \mathsf{if}\ (x \in_? A)\ (f\ x)\ x, \mathsf{restriction\_makes\_map} : \mathsf{map}\ S\ S) : \mathsf{map}\ S\ S$

▷ Discharge x, f

▷ Construct by refinement restriction_forms_map : restriction_fun.is_map
(refl, $\in$, $\neg$, $\in_?$, if, $=$, or_not, case_elim, map)

▷ Define restriction

= $(\mathsf{restriction\_fun}, \mathsf{restriction\_forms\_map} : \mathsf{map}\ (\mathsf{map}\ S\ S)\ (\mathsf{map}\ S\ S))$

: $\mathsf{map}\ (\mathsf{map}\ S\ S)\ (\mathsf{map}\ S\ S)$

▷ Discharge A

▷ Allow $\restriction$ to be written midfix

▷ Define $\restriction = [\lambda f : \mathsf{map}\ S\ S]\ [\lambda A : \mathsf{dsubset}\ S]\ A.\mathsf{restriction}\ f$

$\quad : (\mathsf{map}\ S\ S) \to (\mathsf{dsubset}\ S) \to \mathsf{map}\ S\ S$

▷ Discharge S

B.6.2.2 Restricting all of a subset of mappings

▷ Explicitly overload the identifier $\restriction$

▷ Introduce S

▷ Define $\restriction$

$\quad = [\lambda X : \mathsf{subset}\ (\mathsf{map}\ S\ S)]\ [\lambda A : \mathsf{dsubset}\ S]\ (\mathsf{restriction}\ A).\mathsf{apply\_across}\ X$

$\quad : (\mathsf{subset}\ (\mathsf{map}\ S\ S)) \to (\mathsf{dsubset}\ S) \to \mathsf{subset}\ (\mathsf{map}\ S\ S)$

▷ Discharge S

B.6.2.3 Results concerning restricted mappings

▷ Introduce S; $A : \mathsf{dsubset}\ S$ and $f_1, f_2, f : \mathsf{map}\ S\ S$

▷ Prove $\mathsf{REST}_1 : f.(=\restriction A)\ (f \restriction A)$

$\quad = [\lambda x : A]\ (\mathsf{IF}_1\ (f\ x)\ x\ (x \in_?\ A)\ x.\mathsf{ev}).\mathsf{symm}$

▷ Prove $\mathsf{REST}_2 : \{\forall x \mid S\}\ (x \notin A) \to (f \restriction A\ x) = x$

$\quad = [\lambda x \mid S]\ [\lambda H : x \notin A]\ \mathsf{IF}_0\ (f\ x)\ x\ (x \in_?\ A)\ H$

▷ Suppose $H : f \in (\mathsf{map\_from}\ A)$ and introduce $x : S$

▷ Prove subresult $\mathsf{C}_{1\text{-}1} : (x \in A) \to (f\ x) = (f \restriction A\ x)$

$\quad = [\lambda H' : x \in A]\ \mathsf{REST}_1\ H'$

▷ Suppose $H' : x \notin A$

▷ Prove subresult $\mathsf{Q}_1 : (f\ x) = x$

$\quad = \mathsf{case}\ (H\ x)\ ([\lambda X : x \in A]\ \mathsf{ex\_falso}\ ((f\ x) = x)\ (H'\ X))\ ([\lambda X : (f\ x) = x]\ X)$

▷ Prove subresult $\mathsf{Q}_2 : x = (f \restriction A\ x)$

$\quad = (\mathsf{REST}_2\ H').\mathsf{symm}$

▷ Prove subresult $\mathsf{C}_{1\text{-}2} : (f\ x) = (f \restriction A\ x)$

$\quad = \mathsf{Q}_1.\mathsf{tran}\ \mathsf{Q}_2$

▷ Discharge H'

▷ Prove subresult $\mathsf{P}_1 : (f\ x) = (f{\restriction}A\ x)$
$= \mathsf{case}\ (x \in_?\ A)\ \mathsf{C}_{1\text{-}1}\ \mathsf{C}_{1\text{-}2}$

▷ Discharge x, H

▷ Suppose $H : f = (f{\restriction}A)$ and introduce $x : S$

▷ Prove subresult $\mathsf{C}_{2\text{-}1} : (x \in A) \to (x \in A) \vee ((f\ x) = x)$
$= \mathsf{inl}|(x \in A)|((f\ x) = x)$

▷ Prove subresult $\mathsf{C}_{2\text{-}2} : (x \notin A) \to (x \in A) \vee ((f_{\cdot\mathbf{1}}\ x) = x)$
$= [\lambda H' : x \notin A]\ \mathsf{inr}|(x \in A)\ ((H\ x).\mathsf{tran}\ (\mathsf{REST}_2\ H'))$

▷ Prove subresult $\mathsf{P}_2 : (x \in A) \vee ((f\ x) = x)$
$= \mathsf{case}\ (x \in_?\ A)\ \mathsf{C}_{2\text{-}1}\ \mathsf{C}_{2\text{-}2}$

▷ Discharge x, H

▷ Prove $\mathsf{REST}_3 : (f \in (\mathsf{map\_from}\ A)) \leftrightarrow (f = (f{\restriction}A))$
$= \mathsf{pair}\ \mathsf{P}_1\ \mathsf{P}_2$

▷ Discharge $\mathsf{P}_2, \mathsf{C}_{2\text{-}2}, \mathsf{C}_{2\text{-}1}, \mathsf{P}_1, \mathsf{C}_{1\text{-}2}, \mathsf{Q}_2, \mathsf{Q}_1, \mathsf{C}_{1\text{-}1}, f$

▷ Suppose $Q : f_1.(={\restriction}A)\ f_2$ and introduce $x : S$

▷ Prove subresult $\mathsf{C}_1 : (x \in A) \to (f_1{\restriction}A\ x) = (f_2{\restriction}A\ x)$
$= [\lambda H : x \in A]\ (\mathsf{REST}_1\ f_1\ H).\mathsf{symm.tran}\ ((Q\ H).\mathsf{tran}\ (\mathsf{REST}_1\ f_2\ H))$

▷ Prove subresult $\mathsf{C}_2 : (x \notin A) \to (f_1{\restriction}A\ x) = (f_2{\restriction}A\ x)$
$= [\lambda H : x \notin A]\ (\mathsf{REST}_2\ f_1\ H).\mathsf{tran}\ (\mathsf{REST}_2\ f_2\ H).\mathsf{symm}$

▷ Prove subresult $\mathsf{P}_1 : (f_1{\restriction}A\ x) = (f_2{\restriction}A\ x)$
$= \mathsf{case}\ (x \in_?\ A)\ \mathsf{C}_1\ \mathsf{C}_2$

▷ Discharge x, Q

▷ Suppose $Q : (f_1{\restriction}A) = (f_2{\restriction}A)$

▷ Prove subresult $\mathsf{Q'} : (f_1{\restriction}A).(={\restriction}A)\ (f_2{\restriction}A)$
$= [\lambda x : A]\ Q\ x$

▷ Prove subresult $\mathsf{P}_2 : f_1.(={\restriction}A)\ f_2$
$= (\mathsf{REST}_1\ f_1).\mathsf{tran}\ (\mathsf{Q'}.\mathsf{tran}\ (\mathsf{REST}_1\ f_2).\mathsf{symm})$

▷ Discharge Q

▷ Prove $\mathsf{REST}_4 : (f_1.(={\restriction}A)\ f_2) \leftrightarrow ((f_1{\restriction}A) = (f_2{\restriction}A))$
$= \mathsf{pair}\ \mathsf{P}_1\ \mathsf{P}_2$

▷ Discharge $\mathsf{P}_2, \mathsf{Q'}, \mathsf{P}_1, \mathsf{C}_2, \mathsf{C}_1, f_2, f_1$

▷ Introduce $f : \mathsf{map}\ S\ S$

▷ Prove subresult REST_5' : $(f\restriction A) = ((f\restriction A)\restriction A)$
$= [\lambda x : S]$ case $(x \in_? A)$ $([\lambda H : x \in A]$ REST_1 $(f\restriction A)$ $H)$
$([\lambda H : x \notin A]$ $(\mathsf{REST}_2$ f $H)$.tran $(\mathsf{REST}_2$ $(f\restriction A)$ $H)$.symm)

▷ Prove REST_5 : $(f\restriction A) \in$ (map_from A)
$= (\mathsf{REST}_3$ $(f\restriction A))$.snd REST_5'

▷ Discharge REST_5', f, A, S

B.6.3 The permutation group

B.6.3.1 The group of set permutations

▷ Introduce S : set

▷ Define id_perm = (identity|S, identity_is_iso S : iso S S) : iso S S

▷ Define o_perm = ($[\lambda f, g :$ iso S $S]$ $(f \circ g,$ compose_isos_is_iso f g : iso S $S)$,
$[\lambda f_1, f_2 \,|$ iso S $S]$ $[\lambda Q f : f_1 = f_2]$ $[\lambda g_1, g_2 \,|$ iso S $S]$ $[\lambda Q g : g_1 = g_2]$
$\circ$.resp$_2$ Qf Qg : map$_2$ (iso S S) (iso S S) (iso S S))
: map$_2$ (iso S S) (iso S S) (iso S S)

▷ Prove perm_forms_group : group_axioms|(iso S S) id_perm o_perm $^{-1}$
$=$ pair$_3$ $([\lambda f, g, h :$ iso S $S]$ compose_assoc f g $h)$
(pair $([\lambda f :$ iso S $S]$ f.compose_identity) $([\lambda f :$ iso S $S]$ identity_compose $f))$
$([\lambda f :$ iso S $S]$ iso_compose_inverse $f)$

▷ Define perm
$=$ ((iso S S).as_a_set, id_perm, o_perm, $^{-1}|S|S$, perm_forms_group : group)
: group

▷ Discharge S

▷ Define coercion iso_perm $= [\lambda S]$ $[\lambda f :$ perm $S]$ f : $\{\Pi S\}$ (perm S) $\to$ iso S S

▷ Define coercion perm_iso $= [\lambda S]$ $[\lambda f :$ iso S $S]$ f : $\{\Pi S\}$ (iso S S) $\to$ perm S

▷ Define coercion rep_iso $= [\lambda S]$ $[\lambda A \,|$ subgroup (perm S)] $[\lambda f : A]$
iso_perm (rep f) : $\{\Pi S\}$ $\{\Pi A \,|$ subgroup (perm S)$\}$ $A \to$ iso S S

B.6.3.2 Extension of some previous equipment to permutations

▷ Define $\mathsf{as\_a\_subset\_of\_perm} = [\lambda S]\ [\lambda A : \mathsf{subset}\ (\mathsf{map}\ S\ S)]$

$A.\mathsf{as\_a\_subset\_of}\ (\mathsf{iso}\ S\ S) : \{\Pi S\}\ (\mathsf{subset}\ (\mathsf{map}\ S\ S)) \to \mathsf{subset}\ (\mathsf{perm}\ S)$

▷ Define $\mathsf{as\_a\_subset\_of\_maps}$

$= [\lambda S]\ [\lambda G : \mathsf{subset}\ (\mathsf{perm}\ S)]\ (\mathsf{rep\_map}\ (\mathsf{iso}\ S\ S)).\mathsf{apply\_across}\ G$

$: \{\Pi S\}\ (\mathsf{subset}\ (\mathsf{perm}\ S)) \to \mathsf{subset}\ (\mathsf{map}\ S\ S)$

▷ Prove $\mathsf{rewrite\_o} : \{\forall S\}\ \{\forall f, g : \mathsf{perm}\ S\}\ \{\forall x : S\}\ (f \circ g\ x) = (f\ (g\ x))$
$= [\lambda S]\ [\lambda f, g : \mathsf{perm}\ S]\ [\lambda x : S]\ \mathsf{refl}\ (f \circ g\ x)$

B.6.3.3 Restriction of permutations

▷ Introduce S

▷ Define $\mathsf{restriction\_perm} = [\lambda A : \mathsf{dsubset}\ S]\ (\mathsf{restriction}\ A) \circ$

$(\mathsf{rep\_map}\ (\mathsf{iso}\ S\ S)) : (\mathsf{dsubset}\ S) \to \mathsf{map}\ (\mathsf{iso}\ S\ S)\ (\mathsf{map}\ S\ S)$

▷ Explicitly overload the identifier $\restriction$

▷ Introduce $X : \mathsf{subset}\ (\mathsf{perm}\ S)$ and $A : \mathsf{dsubset}\ S$

▷ Define $\restriction = (\mathsf{restriction\_perm}\ A).\mathsf{apply\_across}\ X : \mathsf{subset}\ (\mathsf{map}\ S\ S)$

▷ Prove $\mathsf{restrict\_perm\_reformulation} : \restriction = (X.\mathsf{as\_a\_subset\_of\_maps}\restriction A)$
$= \mathsf{AA}_3\ X\ (\mathsf{rep\_map}\ (\mathsf{iso}\ S\ S))\ (\mathsf{restriction}\ A)$

▷ Discharge A, X, S

B.6.3.4 Permutations of subsets

▷ Introduce S and $A : \mathsf{subset}\ S$

▷ Define $\mathsf{Perm\_subset} = (\mathsf{map\_from}\ A).\mathsf{as\_a\_subset\_of\_perm} : \mathsf{subset}\ (\mathsf{perm}\ S)$

▷ Let coercion $\mathsf{Perm\_rep} = [\lambda f : \mathsf{Perm\_subset}]\ f.\mathsf{rep} : \mathsf{Perm\_subset} \to \mathsf{perm}\ S$

▷ Introduce $f_1, f_2, f : \mathsf{Perm\_subset}$ and $x : S$

▷ Prove subresult $\mathsf{id\_in\_Perm} : (x \in A) \vee (\mathsf{id}.\mathsf{fixes}\ x)$
$= \mathsf{inr\_is\_true}\ (x \in A)\ (\mathsf{refl}\ x)$

▷ Prove subresult $\mathsf{o\_C}_1 : (x \in A) \to (x \in A) \vee ((f_1 \circ f_2).\mathsf{fixes}\ x)$
$= [\lambda H_1 : x \in A]\ \mathsf{inl\_is\_true}\ H_1\ ((f_1 \circ f_2).\mathsf{fixes}\ x)$

▷ Suppose H_1 : f_2.fixes x

▷ Prove subresult o_C$_{2\text{-}1}$: $(x \in A) \to (x \in A) \vee ((f_1 \circ f_2)$.fixes $x)$
= $[\lambda H_2 : x \in A]$ inl_is_true H_2 $((f_1 \circ f_2)$.fixes $x)$

▷ Suppose H_2 : f_1.fixes x

▷ Prove subresult o_Q$_1$: $(f_1 \circ f_2\ x) = (f_1\ (f_2\ x))$
= rewrite_o f_1 f_2 x

▷ Prove subresult o_Q$_2$: $(f_1\ (f_2\ x)) = (f_1\ x)$
= f_1.resp H_1

▷ Prove subresult o_C$_{2\text{-}2}$: $(x \in A) \vee ((f_1 \circ f_2)$.fixes $x)$
= inr_is_true $(x \in A)$ (o_Q$_1$.tran (o_Q$_2$.tran H_2))

▷ Discharge H_2

▷ Prove subresult o_C$_2$: $(x \in A) \vee ((f_1 \circ f_2)$.fixes $x)$
= case (f_1.ev x) o_C$_{2\text{-}1}$ o_C$_{2\text{-}2}$

▷ Discharge H_1

▷ Prove subresult o_in_Perm : $(x \in A) \vee ((f_1 \circ f_2)$.fixes $x)$
= case (f_2.ev x) o_C$_1$ o_C$_2$

▷ Prove subresult inv_C$_1$: $(x \in A) \to (x \in A) \vee (f^{-1}$.fixes $x)$
= $[\lambda H : x \in A]$ inl_is_true H (f^{-1}.fixes x)

▷ Suppose H : f.fixes x

▷ Prove subresult inv_Q$_1$: $(f^{-1}\ x) = (f^{-1}\ (f\ x))$
= f^{-1}.resp H.symm

▷ Prove subresult inv_Q$_2$: $(f^{-1}\ (f\ x)) = (f^{-1} \circ f\ x)$
= (rewrite_o f^{-1} f x).symm

▷ Prove subresult inv_Q$_3$: $(f^{-1} \circ f\ x) = x$
= inv_o f x

▷ Prove subresult inv_C$_2$: $(x \in A) \vee (f^{-1}$.fixes $x)$
= inr_is_true $(x \in A)$ (inv_Q$_1$.tran (inv_Q$_2$.tran inv_Q$_3$))

▷ Discharge H

▷ Prove subresult inv_in_Perm : $(x \in A) \vee (f^{-1}$.fixes $x)$
= case (f.ev x) inv_C$_1$ inv_C$_2$

▷ Discharge x, f, f_2, f_1

▷ Prove Perm_forms_subgroup : subgroup_axioms Perm_subset
= pair$_3$ id_in_Perm o_in_Perm inv_in_Perm

▷ Define $\mathsf{Perm} = (\mathsf{Perm\_subset}, \mathsf{Perm\_forms\_subgroup} : \mathsf{subgroup}\ (\mathsf{perm}\ S))$
$: \mathsf{subgroup}\ (\mathsf{perm}\ S)$

▷ Discharge $\mathsf{inv\_in\_Perm}, \mathsf{inv\_C}_2, \mathsf{inv\_Q}_3, \mathsf{inv\_Q}_2, \mathsf{inv\_Q}_1, \mathsf{inv\_C}_1, \mathsf{o\_in\_Perm}, \mathsf{o\_C}_2,$
$\mathsf{o\_C}_{2\text{-}2}, \mathsf{o\_Q}_2, \mathsf{o\_Q}_1, \mathsf{o\_C}_{2\text{-}1}, \mathsf{o\_C}_1, \mathsf{id\_in\_Perm}, \mathsf{Perm\_rep}, A, S$

B.6.3.5 Results concerning permutations of subsets

▷ Introduce S; A, B; g : Perm A and $x : A$

▷ Prove subresult $\mathsf{C}_1 : ((g\ x) \in A) \to (g\ x) \in A$
$= [\lambda H_1 : (g\ x) \in A]\ H_1$

▷ Suppose $H_2 : (g\ (g\ x)) = (g\ x)$

▷ Prove subresult $\mathsf{Q}_1 : ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ (g\ x)) = (((g^{-1} \circ g))_{\cdot\mathbf{1}\cdot\mathbf{1}}\ (g\ x))$
$= (\mathsf{inv\_o}\ g\ (g\ x)).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_2 : (g^{-1} \circ g\ (g\ x)) = (g^{-1}\ (g\ (g\ x)))$
$= \mathsf{rewrite\_o}\ g^{-1}\ g\ (g\ x)$

▷ Prove subresult $\mathsf{Q}_3 : (g^{-1}\ (g\ (g\ x))) = (g^{-1}\ (g\ x))$
$= g^{-1}.\mathsf{resp}\ H_2$

▷ Prove subresult $\mathsf{Q}_4 : (((g^{-1} \circ g))_{\cdot\mathbf{1}\cdot\mathbf{1}}\ x) = ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ x)$
$= \mathsf{inv\_o}\ g\ x$

▷ Prove subresult $\mathsf{Q}_5 : g.\mathsf{fixes}\ x$
$= (\mathsf{Q}_1.\mathsf{tran}\ \mathsf{Q}_2).\mathsf{tran}\ (\mathsf{Q}_3.\mathsf{tran}\ \mathsf{Q}_4)$

▷ Prove subresult $\mathsf{C}_2 : (g\ x) \in A$
$= \mathsf{eq\_closed}\ \mathsf{Q}_5.\mathsf{symm}\ x.\mathsf{ev}$

▷ Discharge H_2

▷ Prove $\mathsf{PERMc} : (g\ x) \in A$
$= \mathsf{case}\ (g.\mathsf{ev}\ (g\ x))\ \mathsf{C}_1\ \mathsf{C}_2$

▷ Discharge $\mathsf{C}_2, \mathsf{Q}_5, \mathsf{Q}_4, \mathsf{Q}_3, \mathsf{Q}_2, \mathsf{Q}_1, \mathsf{C}_1, x, g$

▷ Introduce g : Perm A and $x : S$

▷ Prove subresult $\mathsf{P}_1 : (x \in A) \to (g\ x) \in A$
$= [\lambda H : x \in A]\ \mathsf{PERMc}\ g\ H$

▷ Suppose $H : (g\ x) \in A$

▷ Prove subresult $\mathsf{P}_2 : (g^{-1}\ (g\ x)) \in A$
$= \mathsf{PERMc}\ g.\mathsf{inv\_closed}\ H$

▷ Prove subresult Q_1 **:** $(g^{-1}\ (g\ x)) = (g^{-1} \circ g\ x)$
= (rewrite_o g^{-1} g x).symm

▷ Prove subresult Q_2 **:** $(g^{-1} \circ g\ x) = x$
= g.inv_o x

▷ Prove subresult P_3 **:** $x \in A$
= eq_closed (Q_1.tran Q_2) P_2

▷ Discharge H

▷ Prove PERMc_2 **:** $(x \in A) \leftrightarrow ((g\ x) \in A)$
= pair P_1 P_3

▷ Prove $\mathsf{PERM}_1\mathsf{s}$ **:** $(A \subseteq B) \rightarrow (\mathsf{Perm}\ A) \subseteq (\mathsf{Perm}\ B)$
= $[\lambda H : A \subseteq B]\ [\lambda g : \mathsf{Perm}\ A]$ $\mathsf{FROM}_1\mathsf{s}$ H (g.rep.rep.as_el_of (map_from A) g.ev)

▷ Discharge but keep $\mathsf{P}_3, \mathsf{Q}_2, \mathsf{Q}_1, \mathsf{P}_2, \mathsf{P}_1, x, g, B, A$

▷ Prove PERM_1 **:** $(A = B) \rightarrow (\mathsf{Perm}\ A) = (\mathsf{Perm}\ B)$
= $[\lambda Q : A = B]$ pair ($\mathsf{PERM}_1\mathsf{s}$ Q.fst) ($\mathsf{PERM}_1\mathsf{s}$ Q.snd)

▷ Discharge $\mathsf{P}_3, \mathsf{Q}_2, \mathsf{Q}_1, \mathsf{P}_2, \mathsf{P}_1, x, g, B, A, S$

B.7 Finite tuples and vectorspaces

B.7.1 Tuples

B.7.1.1 N-tuples over a set

▷ Allow $^\wedge$ to be written midfix

▷ Allow $\backslash$ to be written midfix

▷ Define $^\wedge = [\lambda S : \mathsf{set}]\ [\lambda n]$ map n S **:** $\mathsf{set} \rightarrow \mathbb{N} \rightarrow \mathsf{set}$

▷ Define $\backslash = [\lambda n \,|\, \mathbb{N}]\ [\lambda S]\ [\lambda \boldsymbol{x} : S^\wedge n]\ [\lambda i]$ $\boldsymbol{x}$.ap i
: $\{\Pi n \,|\, \mathbb{N}\}\ \{\Pi S\}\ (S^\wedge n) \rightarrow n \rightarrow S$

▷ Construct by refinement tuple_forms_subset
: $\{\forall n \,|\, \mathbb{N}\}\ \{\forall S\}\ \{\Pi \boldsymbol{x} : S^\wedge n\}$ subset_axioms $([\lambda x : S]\ \langle \exists i \rangle\ (\boldsymbol{x} \backslash i) = x)$
($\backslash$, tran, $=$, $^\wedge$, set, $\mathbb{N}$)

▷ Define coercion $\mathsf{tuple\_subset} = [\lambda n \,|\, \mathbb{N}]\ [\lambda S]\ [\lambda \boldsymbol{x} : S\hat{}n]$

$(\![\lambda x : S]\ \langle \exists i \rangle\ (\boldsymbol{x}\backslash i) = x, \mathsf{tuple\_forms\_subset}\ \boldsymbol{x} : \mathsf{subset}\ S)$

$: \{\Pi n \,|\, \mathbb{N}\}\ \{\Pi S\}\ (S\hat{}n) \to \mathsf{subset}\ S$

▷ Introduce $n \mid \mathbb{N}$ and S

▷ Define $\mathsf{hd\_fun} = [\lambda \boldsymbol{x} : S\hat{}n{+}1]\ \boldsymbol{x}\backslash(\underline{0}\ n) : (S\hat{}n{+}1) \to S$

▷ Construct by refinement $\mathsf{hd\_forms\_map} : \mathsf{hd\_fun.is\_map}$
$(\underline{0},\ {+}1,\ \hat{},\ =)$

▷ Define $\mathsf{hd} = (\mathsf{hd\_fun}, \mathsf{hd\_forms\_map} : \mathsf{map}\ (S\hat{}n{+}1)\ S) : \mathsf{map}\ (S\hat{}n{+}1)\ S$

▷ Introduce $\boldsymbol{x} : S\hat{}n{+}1$

▷ Define $\mathsf{tl\_aux\_fun} = [\lambda i]\ \boldsymbol{x}\backslash i{+}\underline{1} : n \to S$

▷ Construct by refinement $\mathsf{tl\_aux\_forms\_map} : \mathsf{tl\_aux\_fun}.(\mathsf{is\_map}|n|S)$
$({+}\underline{1},\ {+}1,\ \mathsf{resp},\ =)$

▷ Define $\mathsf{tl\_fun} = (\mathsf{tl\_aux\_fun}, \mathsf{tl\_aux\_forms\_map} : \mathsf{map}\ n\ S) : \mathsf{map}\ n\ S$

▷ Discharge $\boldsymbol{x}$

▷ Construct by refinement $\mathsf{tl\_forms\_map} : \mathsf{tl\_fun.is\_map}$
$({+}\underline{1},\ {+}1,\ \hat{},\ =)$

▷ Define $\mathsf{tl} = (\mathsf{tl\_fun}, \mathsf{tl\_forms\_map} : \mathsf{map}\ (S\hat{}n{+}1)\ (S\hat{}n))$

$: \mathsf{map}\ (S\hat{}n{+}1)\ (S\hat{}n)$

▷ Discharge S, n

B.7.1.2 Tuples made from copies of the same element

▷ Introduce S; $x : S$ and n

▷ Construct by refinement $\mathsf{constant\_forms\_tuple} : ([\lambda\_ : n]\ x).(\mathsf{is\_map}|n|S)$
$(\mathsf{refl},\ =)$

▷ Explicitly overload the identifier $\hat{}$

▷ Define $\hat{} = ([\lambda\_ : n]\ x, \mathsf{constant\_forms\_tuple} : S\hat{}n) : S\hat{}n$

▷ Discharge but keep n, x

▷ Prove $\mathsf{CONST}_1 : (\mathsf{tl}\ (x\hat{}n{+}1)) = (x\hat{}n)$
$= [\lambda\_ : n]\ \mathsf{refl}\ x$

▷ Define $\mathsf{one\_tuple} = x\hat{}1 : S\hat{}1$

▷ Discharge n, x, S

B.7.1.3 Cartesian powers of a subset

▷ Introduce S; A : subset S and n

▷ Prove power_forms_subset : subset_axioms $([\lambda x : S^\wedge n]\ \{\forall i\}\ (x_\backslash i) \in A)$
$= [\lambda x, y \mid S^\wedge n]\ [\lambda Q : x = y]\ [\lambda X : \{\forall i\}\ (x_\backslash i) \in A]\ [\lambda i]$ eq_closed $(Q\ i)\ (X\ i)$

▷ Explicitly overload the identifier $^\wedge$

▷ Define $^\wedge = ([\lambda x : S^\wedge n]\ \{\forall i\}\ (x_\backslash i) \in A, \text{power\_forms\_subset} : \text{subset}\ (S^\wedge n))$
: subset $(S^\wedge n)$

▷ Discharge n, A

▷ Introduce A, B

▷ Prove $\text{POWER}_1\text{s} : (A \subseteq B) \to \{\forall n\}\ (A^\wedge n) \subseteq (B^\wedge n)$
$= [\lambda H : A \subseteq B]\ [\lambda n]\ [\lambda x : A^\wedge n]\ [\lambda i]\ H\ (x.\text{ev}\ i)$

▷ Discharge but keep B, A

▷ Prove $\text{POWER}_1 : (A = B) \to \{\forall n\}\ (A^\wedge n) = (B^\wedge n)$
$= [\lambda Q : A = B]\ [\lambda n]$ pair $(\text{POWER}_1\text{s}\ Q.\text{fst}\ n)\ (\text{POWER}_1\text{s}\ Q.\text{snd}\ n)$

▷ Discharge B, A, S

B.7.1.4 Extending unary and binary maps to work on tuples

▷ Introduce S, T, U | set; n | $\mathbb{N}$; f : map $S\ T$ and $\boldsymbol{x} : S^\wedge n$

▷ Prove zip_forms_tuple : $([\lambda i]\ f\ (\boldsymbol{x}_\backslash i)).(\text{is\_map}|n|T)$
$= [\lambda i_1, i_2 \mid n]\ [\lambda Q : i_1 = i_2]\ f.\text{resp}\ (\boldsymbol{x}.\text{resp}\ Q)$

▷ Define zip_tuple $= ([\lambda i]\ f\ (\boldsymbol{x}_\backslash i), \text{zip\_forms\_tuple} : T^\wedge n) : T^\wedge n$

▷ Discharge $\boldsymbol{x}$

▷ Prove zip_aux_forms_map : zip_tuple.is_map
$= [\lambda \boldsymbol{x}, \boldsymbol{y} \mid S^\wedge n]\ [\lambda Q : \boldsymbol{x} = \boldsymbol{y}]\ [\lambda i]\ f.\text{resp}\ (Q\ i)$

▷ Define zip_aux $= (\text{zip\_tuple}, \text{zip\_aux\_forms\_map} : \text{map}\ (S^\wedge n)\ (T^\wedge n))$
: map $(S^\wedge n)\ (T^\wedge n)$

▷ Discharge f

▷ Prove $\mathsf{zip\_forms\_map} : \mathsf{zip\_aux.is\_map}$
$= [\lambda f_1, f_2 \,|\, \mathsf{map}\ S\ T]\ [\lambda Q : f_1 = f_2]\ [\lambda \boldsymbol{x} : S^\wedge n]\ [\lambda i]\ Q\ (\boldsymbol{x}_\backslash i)$

▷ Define $\mathsf{zip} = (\mathsf{zip\_aux}, \mathsf{zip\_forms\_map} : \mathsf{map}\ (\mathsf{map}\ S\ T)\ (\mathsf{map}\ (S^\wedge n)\ (T^\wedge n)))$
$: \mathsf{map}\ (\mathsf{map}\ S\ T)\ (\mathsf{map}\ (S^\wedge n)\ (T^\wedge n))$

▷ Discharge n

▷ Explicitly overload the identifier $^\wedge$

▷ Define $^\wedge = [\lambda f : \mathsf{map}\ S\ T]\ [\lambda n]\ \mathsf{zip}|n\ f$
$: (\mathsf{map}\ S\ T) \to \{\Pi n\}\ \mathsf{map}\ (S^\wedge n)\ (T^\wedge n)$

▷ Introduce $n \mid \mathbb{N}$; $\boldsymbol{x} : S^\wedge n{+}1$ and $f : \mathsf{map}\ S\ T$

▷ Prove $\mathsf{map\_tail} : (f^\wedge n\ \boldsymbol{x}.\mathsf{tl}) = (f^\wedge n{+}1\ \boldsymbol{x}).\mathsf{tl}$
$= [\lambda i]\ \mathsf{refl}\ (f\ (\boldsymbol{x}_\backslash i{+}\underline{1}))$

▷ Discharge $f, \boldsymbol{x}$

▷ Introduce $f : \mathsf{map}_2\ S\ T\ U$; $\boldsymbol{v} : S^\wedge n$ and $\boldsymbol{x} : T^\wedge n$

▷ Prove $\mathsf{zip}_2\mathsf{\_forms\_tuple} : ([\lambda i]\ f\ (\boldsymbol{v}_\backslash i)\ (\boldsymbol{x}_\backslash i)).(\mathsf{is\_map}|n|U)$
$= [\lambda i_1, i_2 \,|\, n]\ [\lambda Q : i_1 = i_2]\ f.\mathsf{resp}_2\ (\boldsymbol{v}.\mathsf{resp}\ Q)\ (\boldsymbol{x}.\mathsf{resp}\ Q)$

▷ Define $\mathsf{zip}_2\mathsf{\_tuple} = ([\lambda i]\ f\ (\boldsymbol{v}_\backslash i)\ (\boldsymbol{x}_\backslash i), \mathsf{zip}_2\mathsf{\_forms\_tuple} : U^\wedge n) : U^\wedge n$

▷ Discharge $\boldsymbol{x}, \boldsymbol{v}$

▷ Prove $\mathsf{zip}_2\mathsf{\_aux\_forms\_map}_2 : \mathsf{zip}_2\mathsf{\_tuple.is\_map}_2$
$= [\lambda \boldsymbol{v}, \boldsymbol{w} \,|\, S^\wedge n]\ [\lambda Q_1 : \boldsymbol{v} = \boldsymbol{w}]\ [\lambda \boldsymbol{x}, \boldsymbol{y} \,|\, T^\wedge n]\ [\lambda Q_2 : \boldsymbol{x} = \boldsymbol{y}]\ [\lambda i]$
$f.\mathsf{resp}_2\ (Q_1\ i)\ (Q_2\ i)$

▷ Define $\mathsf{zip}_2\mathsf{\_aux}$
$= (\mathsf{zip}_2\mathsf{\_tuple}, \mathsf{zip}_2\mathsf{\_aux\_forms\_map}_2 : \mathsf{map}_2\ (S^\wedge n)\ (T^\wedge n)\ (U^\wedge n))$
$: \mathsf{map}_2\ (S^\wedge n)\ (T^\wedge n)\ (U^\wedge n)$

▷ Discharge f

▷ Prove $\mathsf{zip}_2\mathsf{\_forms\_map} : \mathsf{zip}_2\mathsf{\_aux.is\_map}$
$= [\lambda f_1, f_2 \,|\, \mathsf{map}_2\ S\ T\ U]\ [\lambda Q : f_1 = f_2]\ [\lambda \boldsymbol{v} : S^\wedge n]\ [\lambda \boldsymbol{x} : T^\wedge n]\ [\lambda i]\ Q\ (\boldsymbol{v}_\backslash i)\ (\boldsymbol{x}_\backslash i)$

▷ Define $\mathsf{zip}_2 = (\mathsf{zip}_2\mathsf{\_aux}, \mathsf{zip}_2\mathsf{\_forms\_map} :$
$\mathsf{map}\ (\mathsf{map}_2\ S\ T\ U)\ (\mathsf{map}_2\ (S^\wedge n)\ (T^\wedge n)\ (U^\wedge n)))$
$: \mathsf{map}\ (\mathsf{map}_2\ S\ T\ U)\ (\mathsf{map}_2\ (S^\wedge n)\ (T^\wedge n)\ (U^\wedge n))$

▷ Discharge n

▷ Explicitly overload the identifier $^\wedge$

▷ Define $\mathsf{map_2le} = [\lambda f : \mathsf{map_2}\ S\ T\ U]\ [\lambda n]\ \mathsf{zip_2}|n\ f$

: $(\mathsf{map_2}\ S\ T\ U) \to \{\Pi n\}\ \mathsf{map_2}\ (S\hat{}n)\ (T\hat{}n)\ (U\hat{}n)$

▷ Introduce $n \mid \mathbb{N}$; $\boldsymbol{x} : S\hat{}n{+}1$; $\boldsymbol{y} : T\hat{}n{+}1$; $f :$ $\mathsf{map_2}\ S\ T\ U$ and prove
$\mathsf{map_2\_tail} : (f.\mathsf{map_2le}\ n\ \boldsymbol{x}.\mathsf{tl}\ \boldsymbol{y}.\mathsf{tl}) = (f.\mathsf{map_2le}\ n{+}1\ \boldsymbol{x}\ \boldsymbol{y}).\mathsf{tl}$
$= [\lambda i]\ \mathsf{refl}\ (f\ (\boldsymbol{x}\backslash i{+}\underline{1})\ (\boldsymbol{y}\backslash i{+}\underline{1}))$

▷ Discharge $f, \boldsymbol{y}, \boldsymbol{x}, n, U, T, S$

B.7.1.5 Folding a map across a tuple

▷ Introduce S; $base : S$ and $step :$ $\mathsf{map_2}\ S\ S\ S$

▷ Define $\mathsf{fold\_fun} = \mathsf{nat\_elim}\ ([\lambda n]\ (S\hat{}n) \to S)\ ([\lambda\_ : S\hat{}0]\ base)\ ([\lambda n]$

$[\lambda fold\_fun\_n : (S\hat{}n) \to S]\ [\lambda \boldsymbol{x} : S\hat{}n{+}1]\ step\ (\mathsf{hd}\ \boldsymbol{x})\ (fold\_fun\_n\ (\mathsf{tl}\ \boldsymbol{x})))$

: $\{\Pi n \mid \mathbb{N}\}\ (S\hat{}n) \to S$

▷ Construct by refinement $\mathsf{fold\_forms\_map} : \{\Pi n \mid \mathbb{N}\}\ (\mathsf{fold\_fun}|n).\mathsf{is\_map}$
(tl, ^, +1, resp, hd, $\mathbb{N}$, 0, nat_elim, $\mathsf{resp_2}$, succ, =, fold_fun, is_map, nat, zero, refl)

▷ Define $\mathsf{fold} = [\lambda n \mid \mathbb{N}]\ (\mathsf{fold\_fun}|n, \mathsf{fold\_forms\_map}|n : \mathsf{map}\ (S\hat{}n)\ S)$

: $\{\Pi n \mid \mathbb{N}\}\ \mathsf{map}\ (S\hat{}n)\ S$

▷ Discharge $step, base, S$

B.7.1.6 Linear sum in a group

▷ Define $\sum = [\lambda G \mid \mathsf{abelian\_group}]\ \mathsf{fold}|G\ \mathbf{0}\ +$

: $\{\Pi G \mid \mathsf{abelian\_group}\}\ \{\Pi n \mid \mathbb{N}\}\ \mathsf{map}\ (G\hat{}n)\ G$

B.7.1.7 Results concerning linear sums

▷ Introduce $G \mid \mathsf{abelian\_group}$ and $L :$ subgroup G

▷ Let $\mathsf{p} = [\lambda n]\ (\sum|G\ (\mathbf{0}\hat{}n)) = \mathbf{0} : \mathbb{N} \to \mathsf{prop}$

▷ Prove subresult $\mathsf{P_0} : \mathsf{p}\ 0$
$= \mathsf{refl}|G\ \mathbf{0}$

▷ Introduce $n \mid \mathbb{N}$ and suppose $ih :$ p n

▷ Prove subresult $\mathsf{Q}_1 : (\mathbf{0} + (\sum\ (\mathsf{tl}\ (\mathbf{0}\hat{}n{+}1)))) = (\mathbf{0} + (\sum\ (\mathbf{0}\hat{}n)))$
$= \mathsf{plus}_2|G\ \mathbf{0}\ (\sum.\mathsf{resp}\ (\mathsf{CONST}_1\ \mathbf{0}\ n))$

▷ Prove subresult $\mathsf{Q}_2 : (\mathbf{0} + (\sum|G\ (\mathbf{0}\hat{}n))) = (\sum|G\ (\mathbf{0}\hat{}n))$
$= \mathsf{ze\_plus}\ (\sum|G\ (\mathbf{0}\hat{}n))$

▷ Prove subresult $\mathsf{Ps} : \mathsf{p}\ n{+}1$
$= (\mathsf{Q}_1.\mathsf{tran}\ \mathsf{Q}_2).\mathsf{tran}\ ih$

▷ Discharge ih, n

▷ Prove $\mathsf{LINSUM}_0 : \{\forall n\}\ (\sum|G\ (\mathbf{0}\hat{}n)) = \mathbf{0}$
$= \mathsf{N\_elim}\ \mathsf{p}\ \mathsf{P}_0\ \mathsf{Ps}$

▷ Discharge $\mathsf{Ps}, \mathsf{Q}_2, \mathsf{Q}_1, \mathsf{P}_0, \mathsf{p}$

▷ Let $\mathsf{p} = [\lambda n]\ \{\forall \boldsymbol{x}\,|\,G\hat{}n\}\ (\boldsymbol{x} \in (L\hat{}n)) \to (\sum \boldsymbol{x}) \in L : \mathbb{N} \to \mathsf{prop}$

▷ Prove subresult $\mathsf{P}_0 : \mathsf{p}\ 0$
$= [\lambda \boldsymbol{x} : G\hat{}0]\ [\lambda\_ : \boldsymbol{x} \in (L\hat{}0)]\ (\mathsf{id}|L).\mathsf{ev}$

▷ Introduce $n \mid \mathbb{N}$; suppose $ih : \mathsf{p}\ n$; introduce $\boldsymbol{x} : G\hat{}n{+}1$ and suppose $H : \boldsymbol{x} \in (L\hat{}n{+}1)$

▷ Prove subresult $\mathsf{P}_1 : (\boldsymbol{x}\_(\underline{0}\ n)) \in L$
$= H\ (\underline{0}\ n)$

▷ Prove subresult $\mathsf{P}_2 : (\sum\ (\mathsf{tl}\ \boldsymbol{x})) \in L$
$= ih|(\mathsf{tl}\ \boldsymbol{x})\ ([\lambda i]\ H\ (+\underline{1}\ i))$

▷ Prove subresult $\mathsf{Ps} : (\mathsf{P}_1.\mathsf{o\_closed}\ \mathsf{P}_2) \in L\_\mathbf{1}$
$= (\mathsf{P}_1.\mathsf{o\_closed}\ \mathsf{P}_2).\mathsf{ev}$

▷ Discharge $H, \boldsymbol{x}, ih, n$

▷ Prove $\mathsf{LINSUM}_1 : \{\forall n \,|\, \mathbb{N}\}\ \{\forall \boldsymbol{x}\,|\,G\hat{}n\}\ (\boldsymbol{x} \in (L\hat{}n)) \to (\sum \boldsymbol{x}) \in L$
$= \mathsf{N\_elim}\ \mathsf{p}\ \mathsf{P}_0\ \mathsf{Ps}$

▷ Discharge $\mathsf{Ps}, \mathsf{P}_2, \mathsf{P}_1, \mathsf{P}_0, \mathsf{p}$

▷ Introduce $g : \mathsf{subgroup\_hom}\ L$

▷ Let $\mathsf{p} = [\lambda n]\ \{\forall \boldsymbol{x} : L\hat{}n\}\ (g\ (\sum \boldsymbol{x})) = (\sum\ (g \circ \boldsymbol{x}.\mathsf{rep})) : \mathbb{N} \to \mathsf{prop}$

▷ Prove subresult $\mathsf{P}_0 : \mathsf{p}\ 0$
$= [\lambda\_ : L\hat{}0]\ g.\mathsf{on\_ze}$

▷ Introduce $n \mid \mathbb{N}$; suppose $ih : \mathsf{p}\ n$ and introduce $\boldsymbol{x} : L\hat{}n{+}1$

▷ Prove subresult $\mathsf{P}_1 : (\boldsymbol{x}\_(\underline{0}\ n)) \in L$
$= \boldsymbol{x}.\mathsf{ev}\ (\underline{0}\ n)$

▷ Prove subresult P_2 **:** $\boldsymbol{x}.\mathsf{tl} \in (L\hat{}n)$
$= [\lambda i]\ \boldsymbol{x}.\mathsf{ev}\ (+\underline{1}\ i)$

▷ Prove subresult P_3 **:** $(\sum \boldsymbol{x}.\mathsf{tl}) \in L$
$= \mathsf{LINSUM}_1\ \mathsf{P}_2$

▷ Prove subresult Q_1 **:** $(g\ (\sum \boldsymbol{x})) = ((g\ \boldsymbol{x}.\mathsf{hd}) + (g\ (\sum \boldsymbol{x}.\mathsf{tl})))$
$= g.\mathsf{on\_plus}\ \mathsf{P}_1\ \mathsf{P}_3$

▷ Prove subresult $\mathsf{Q}_{2\text{-}1}$ **:** $(g\ \boldsymbol{x}.\mathsf{hd}) = (g \circ \boldsymbol{x}.\mathsf{rep}).\mathsf{hd}$
$= \mathsf{refl}\ (g\ \boldsymbol{x}.\mathsf{hd})$

▷ Prove subresult $\mathsf{Q}_{2\text{-}2\text{-}1}$ **:** $(g\ (\sum \mathsf{P}_2)) = (\sum (g \circ \mathsf{P}_2.\mathsf{rep}))$
$= ih\ \mathsf{P}_2$

▷ Prove subresult P_4 **:** $(g \circ \boldsymbol{x}.\mathsf{tl}) = (g \circ \boldsymbol{x}.\mathsf{rep}).\mathsf{tl}$
$= [\lambda i]\ \mathsf{refl}\ (g\ (\boldsymbol{x}.\mathsf{tl}\backslash i))$

▷ Prove subresult $\mathsf{Q}_{2\text{-}2\text{-}2}$ **:** $(\sum (g \circ \boldsymbol{x}.\mathsf{tl})) = (\sum (g \circ \boldsymbol{x}.\mathsf{rep}).\mathsf{tl})$
$= \sum.\mathsf{resp}\ \mathsf{P}_4$

▷ Prove subresult $\mathsf{Q}_{2\text{-}2}$ **:** $(g\ (\sum \mathsf{P}_2)) = (\sum (g \circ \boldsymbol{x}.\mathsf{rep}).\mathsf{tl})$
$= \mathsf{Q}_{2\text{-}2\text{-}1}.\mathsf{tran}\ \mathsf{Q}_{2\text{-}2\text{-}2}$

▷ Prove subresult Q_2
: $((g\ \boldsymbol{x}.\mathsf{hd}) + (g\ (\sum \mathsf{P}_2))) = ((g \circ \boldsymbol{x}.\mathsf{rep}).\mathsf{hd} + (\sum (g \circ \boldsymbol{x}.\mathsf{rep}).\mathsf{tl}))$
$= \mathsf{plus\_resp}\ \mathsf{Q}_{2\text{-}1}\ \mathsf{Q}_{2\text{-}2}$

▷ Prove subresult Ps **:** $(g\ (\sum \boldsymbol{x})) = (\sum (g \circ \boldsymbol{x}.\mathsf{rep}))$
$= \mathsf{Q}_1.\mathsf{tran}\ \mathsf{Q}_2$

▷ Discharge $\boldsymbol{x}, ih, n$

▷ Prove LINSUM_2 **:** $\{\forall n \,|\, \mathbb{N}\}\ \{\forall \boldsymbol{x} : L\hat{}n\}\ (g\ (\sum \boldsymbol{x})) = (\sum (g \circ \boldsymbol{x}.\mathsf{rep}))$
$= \mathsf{N\_elim}\ \mathsf{p}\ \mathsf{P}_0\ \mathsf{Ps}$

▷ Discharge $\mathsf{Ps}, \mathsf{Q}_2, \mathsf{Q}_{2\text{-}2}, \mathsf{Q}_{2\text{-}2\text{-}2}, \mathsf{P}_4, \mathsf{Q}_{2\text{-}2\text{-}1}, \mathsf{Q}_{2\text{-}1}, \mathsf{Q}_1, \mathsf{P}_3, \mathsf{P}_2, \mathsf{P}_1, \mathsf{P}_0, \mathsf{p}, g, L, G$

▷ Introduce G_1, G_2, G_3 | $\mathsf{abelian\_group}$

▷ Introduce f **:** $\mathsf{map}\ G_1\ G_2$ and suppose H
: $\{\forall x, y : G_1\}\ (f\ (x + y)) = ((f\ x) + (f\ y))$

▷ We want to prove LINSUM_3
: $\{\forall n \,|\, \mathbb{N}\}\ \{\forall \boldsymbol{x} : G_1\hat{}n\}\ (f\ (\sum \boldsymbol{x})) = (\sum (f\hat{}n\ \boldsymbol{x}))$

▷ Let $\mathsf{p} = [\lambda n]\ \{\Pi \boldsymbol{x} : G_1\hat{}n\}\ (f\ (\sum \boldsymbol{x})) = (\sum (f\hat{}n\ \boldsymbol{x})) : \mathbb{N} \to \mathsf{prop}$

▷ Prove subresult Q_0 **:** $(f\ (\mathbf{0} + \mathbf{0})) = (f\ \mathbf{0})$
$= f.\mathsf{resp}\ (\mathsf{ze\_plus}\ \mathbf{0})$

▷ Prove subresult P_0 **:** $\mathsf{p}\ 0$
$= [\lambda\_ : {G_1}\hat{}0]\ \mathsf{GROUP}_4\ ((H\ \mathbf{0}\ \mathbf{0}).\mathsf{symm.tran}\ \mathsf{Q}_0)$

▷ Introduce n and suppose ih **:** $\mathsf{p}\ n$

▷ We want to prove subresult Ps **:** $\mathsf{p}\ n{+}1$

▷ Introduce $\boldsymbol{x}$ **:** ${G_1}\hat{}n{+}1$

▷ Prove subresult Q_1 **:** $(f\ (\boldsymbol{x}.\mathsf{hd} + (\sum\ \boldsymbol{x}.\mathsf{tl}))) = ((f\ \boldsymbol{x}.\mathsf{hd}) + (f\ (\sum\ \boldsymbol{x}.\mathsf{tl})))$
$= H\ \boldsymbol{x}.\mathsf{hd}\ (\sum\ \boldsymbol{x}.\mathsf{tl})$

▷ Prove subresult $\mathsf{Q}_{2\text{-}1}$ **:** $(f\ (\sum\ \boldsymbol{x}.\mathsf{tl})) = (\sum\ (f\hat{}n\ \boldsymbol{x}.\mathsf{tl}))$
$= ih\ \boldsymbol{x}.\mathsf{tl}$

▷ Prove subresult $\mathsf{Q}_{2\text{-}2}$ **:** $(\sum\ (f\hat{}n\ \boldsymbol{x}.\mathsf{tl})) = (\sum\ (f\hat{}n{+}1\ \boldsymbol{x}).\mathsf{tl})$
$= \sum.\mathsf{resp}\ (\mathsf{map\_tail}\ \boldsymbol{x}\ f)$

▷ Prove subresult Q_2
: $((f\ \boldsymbol{x}.\mathsf{hd}) + (f\ (\sum\ \boldsymbol{x}.\mathsf{tl}))) = ((f\ \boldsymbol{x}.\mathsf{hd}) + (\sum\ (f\hat{}n{+}1\ \boldsymbol{x}).\mathsf{tl}))$
$= (f\ \boldsymbol{x}.\mathsf{hd}).\mathsf{plus}_2\ (\mathsf{Q}_{2\text{-}1}.\mathsf{tran}\ \mathsf{Q}_{2\text{-}2})$

▷ Discharge to prove subresult as claimed Ps **:** $\mathsf{p}\ n{+}1$
using
$\mathsf{Q}_1.\mathsf{tran}\ \mathsf{Q}_2$ **:** $(f\ (\boldsymbol{x}.\mathsf{hd} + (\sum\ \boldsymbol{x}.\mathsf{tl}))) = ((f\ \boldsymbol{x}.\mathsf{hd}) + (\sum\ (f\hat{}n{+}1\ \boldsymbol{x}).\mathsf{tl}))$

▷ Discharge $\boldsymbol{x}$

▷ Discharge ih, n

▷ Prove as claimed LINSUM_3
: $\{\forall n \,|\, \mathbb{N}\}\ \{\forall \boldsymbol{x} : {G_1}\hat{}n\}\ (f\ (\sum\ \boldsymbol{x})) = (\sum\ (f\hat{}n\ \boldsymbol{x}))$
$= \mathsf{N\_elim}\ \mathsf{p}\ \mathsf{P}_0\ \mathsf{Ps}$

▷ Discharge $\mathsf{Ps}, \mathsf{Q}_2, \mathsf{Q}_{2\text{-}2}, \mathsf{Q}_{2\text{-}1}, \mathsf{Q}_1, \mathsf{P}_0, \mathsf{Q}_0, \mathsf{p}, H, f$

▷ Introduce f **:** $\mathsf{map}_2\ G_1\ G_2\ G_3$

▷ Suppose H
: $\{\forall x, x' : G_1\}\ \{\forall y, y' : G_2\}\ (f\ (x + x')\ (y + y')) = ((f\ x\ y) + (f\ x'\ y'))$

▷ We want to prove LINSUM_4 **:** $\{\forall n \,|\, \mathbb{N}\}\ \{\forall \boldsymbol{x} : {G_1}\hat{}n\}\ \{\forall \boldsymbol{y} : {G_2}\hat{}n\}$
$(f\ (\sum\ \boldsymbol{x})\ (\sum\ \boldsymbol{y})) = (\sum\ (f.\mathsf{map}_2\mathsf{le}\ n\ \boldsymbol{x}\ \boldsymbol{y}))$

▷ Let $\mathsf{p} = [\lambda n]\ \{\Pi \boldsymbol{x} : {G_1}\hat{}n\}\ \{\Pi \boldsymbol{y} : {G_2}\hat{}n\}$
$(f\ (\sum\ \boldsymbol{x})\ (\sum\ \boldsymbol{y})) = (\sum\ (f.\mathsf{map}_2\mathsf{le}\ n\ \boldsymbol{x}\ \boldsymbol{y}))$ **:** $\mathbb{N} \to \mathsf{prop}$

▷ Prove subresult Q_0 **:** $(f\ (\mathbf{0} + \mathbf{0})\ (\mathbf{0} + \mathbf{0})) = (f\ \mathbf{0}\ \mathbf{0})$
$= f.\mathsf{resp}_2\ (\mathsf{ze\_plus}\ \mathbf{0})\ (\mathsf{ze\_plus}\ \mathbf{0})$

▷ Prove subresult P_0 : p 0
= [λ_ : G_1^0] [λ_ : G_2^0] GROUP$_4$ ((H **0 0 0 0**).symm.tran Q_0)

▷ Introduce n and suppose ih : p n

▷ We want to prove subresult Ps : p n+1

▷ Introduce $\boldsymbol{x}$: G_1^n+1 and $\boldsymbol{y}$: G_2^n+1

▷ Prove subresult Q_1 : (f ($\boldsymbol{x}$.hd + ($\sum$ $\boldsymbol{x}$.tl)) ($\boldsymbol{y}$.hd + ($\sum$ $\boldsymbol{y}$.tl))) =
((f $\boldsymbol{x}$.hd $\boldsymbol{y}$.hd) + (f ($\sum$ $\boldsymbol{x}$.tl) ($\sum$ $\boldsymbol{y}$.tl)))
= H $\boldsymbol{x}$.hd ($\sum$ $\boldsymbol{x}$.tl) $\boldsymbol{y}$.hd ($\sum$ $\boldsymbol{y}$.tl)

▷ Prove subresult $\mathsf{Q}_{2\text{-}1}$: (f ($\sum$ $\boldsymbol{x}$.tl) ($\sum$ $\boldsymbol{y}$.tl)) = ($\sum$ (f.map$_2$le n $\boldsymbol{x}$.tl $\boldsymbol{y}$.tl))
= ih $\boldsymbol{x}$.tl $\boldsymbol{y}$.tl

▷ Prove subresult $\mathsf{Q}_{2\text{-}2}$
: ($\sum$ (f.map$_2$le n $\boldsymbol{x}$.tl $\boldsymbol{y}$.tl)) = ($\sum$ (f.map$_2$le n+1 $\boldsymbol{x}$ $\boldsymbol{y}$).tl)
= $\sum$.resp (map$_2$_tail $\boldsymbol{x}$ $\boldsymbol{y}$ f)

▷ Prove subresult Q_2 : ((f $\boldsymbol{x}$.hd $\boldsymbol{y}$.hd) + (f ($\sum$ $\boldsymbol{x}$.tl) ($\sum$ $\boldsymbol{y}$.tl))) =
((f $\boldsymbol{x}$.hd $\boldsymbol{y}$.hd) + ($\sum$ (f.map$_2$le n+1 $\boldsymbol{x}$ $\boldsymbol{y}$).tl))
= (f $\boldsymbol{x}$.hd $\boldsymbol{y}$.hd).plus$_2$ ($\mathsf{Q}_{2\text{-}1}$.tran $\mathsf{Q}_{2\text{-}2}$)

▷ Discharge to prove subresult as claimed Ps : p n+1
using
Q_1.tran Q_2 : (f ($\boldsymbol{x}$.hd + ($\sum$ $\boldsymbol{x}$.tl)) ($\boldsymbol{y}$.hd + ($\sum$ $\boldsymbol{y}$.tl))) =
((f $\boldsymbol{x}$.hd $\boldsymbol{y}$.hd) + ($\sum$ (f.map$_2$le n+1 $\boldsymbol{x}$ $\boldsymbol{y}$).tl))

▷ Discharge $\boldsymbol{y}, \boldsymbol{x}$

▷ Discharge ih, n

▷ Prove as claimed LINSUM$_4$: {$\forall n \,|\, \mathbb{N}$} {$\forall \boldsymbol{x} : G_1$^$n$} {$\forall \boldsymbol{y} : G_2$^$n$}
(f ($\sum$ $\boldsymbol{x}$) ($\sum$ $\boldsymbol{y}$)) = ($\sum$ (f.map$_2$le n $\boldsymbol{x}$ $\boldsymbol{y}$))
= N_elim p P_0 Ps

▷ Discharge Ps, Q_2, $\mathsf{Q}_{2\text{-}2}$, $\mathsf{Q}_{2\text{-}1}$, Q_1, P_0, Q_0, p, H, f, G_3, G_2, G_1

B.7.2 Vectorspaces

B.7.2.1 Vectorspaces over a field

▷ Allow -space to be written postfix

▷ Define $\mathsf{space\_axioms} = [\lambda F]\ [\lambda V \mid \mathsf{abelian\_group}]\ [\lambda sc : \mathsf{map}_2\ F\ V\ V]$

$(\{\forall x, y : F\}\ \{\forall v : V\}\ ((x + y).sc\ v) = ((x.sc\ v) + (y.sc\ v))) \wedge$

$(\{\forall x : F\}\ \{\forall v, w : V\}\ (x.sc\ (v + w)) = ((x.sc\ v) + (x.sc\ w)))) \wedge$

$((\{\forall x, y : F\}\ \{\forall v : V\}\ ((x \times y).sc\ v) = (x.sc\ (y.sc\ v))) \wedge$

$(\{\forall v : V\}\ (\mathbf{1}.sc\ v) = v))$

$: \{\Pi F\}\ \{\Pi V \mid \mathsf{abelian\_group}\}\ (\mathsf{map}_2\ F\ V\ V) \to \mathsf{prop}$

▷ Define $\mathsf{-space} = [\lambda F : \mathsf{field}]\ \langle \Sigma V : \mathsf{abelian\_group} \rangle\ \langle \Sigma sc : \mathsf{map}_2\ F\ V\ V \rangle$

$\mathsf{space\_axioms}\ sc : \mathsf{field} \to \mathsf{Type}_1$

▷ Define coercion $\mathsf{spg} = [\lambda F]\ [\lambda V : F\text{-}\mathsf{space}]\ V_{.\mathbf{1}}$

$: \{\Pi F\}\ F\text{-}\mathsf{space} \to \mathsf{abelian\_group}$

▷ Prove $\mathsf{field\_forms\_space} : \{\Pi F : \mathsf{field}\}\ \mathsf{space\_axioms}|F|F\ \times$
$= [\lambda F : \mathsf{field}]$
$\mathsf{pair}\ (\mathsf{pair}\ (\mathsf{plus\_times}|F)\ (\mathsf{times\_plus}|F))\ (\mathsf{pair}\ (\mathsf{times\_assoc}|F)\ (\mathsf{un\_times}|F))$

▷ Define coercion $\mathsf{as\_space\_over\_itself}$

$= [\lambda F : \mathsf{field}]\ (F, \times|F, \mathsf{field\_forms\_space}\ F : F\text{-}\mathsf{space}) : \{\Pi F : \mathsf{field}\}\ F\text{-}\mathsf{space}$

▷ Define coercion $\mathsf{subset\_into\_subspace} = [\lambda F]\ [\lambda K : \mathsf{subset}\ F]\ K$

$: \{\Pi F\}\ (\mathsf{subset}\ F) \to \mathsf{subset}\ F.\mathsf{as\_space\_over\_itself}$

▷ Allow $*$ to be written infix

▷ Introduce F

▷ Unless otherwise specified, by default $V : F\text{-}\mathsf{space}$

▷ Introduce $V \mid F\text{-}\mathsf{space}$

▷ Define $* = V_{.\mathbf{2}\cdot\mathbf{1}} : \mathsf{map}_2\ F\ V\ V$

▷ Introduce $x_1, x_2 \mid F$; suppose $Qx : x_1 = x_2$; introduce $x, y : F$; $v_1, v_2 \mid V$;

suppose $Qv : v_1 = v_2$ and introduce $v, w : V$

▷ Prove $\mathsf{sc\_resp} : (x_1 * v_1) = (x_2 * v_2)$
$= \mathsf{resp}_2\ *\ Qx\ Qv$

▷ Prove $\mathsf{sc}_1 : (x_1 * v) = (x_2 * v)$
$= \mathsf{resps}_1\ *\ Qx\ v$

▷ Prove $\mathsf{sc}_2 : (x * v_1) = (x * v_2)$
$= \mathsf{resps}_2\ *\ x\ Qv$

▷ Prove plus_sc : $((x + y) * v) = ((x * v) + (y * v))$
$= V_{.\mathbf{2}.\mathbf{2}}$.fst.fst $x\ y\ v$

▷ Prove sc_plus : $(x * (v + w)) = ((x * v) + (x * w))$
$= V_{.\mathbf{2}.\mathbf{2}}$.fst.snd $x\ v\ w$

▷ Prove times_sc : $((x \times y) * v) = (x * (y * v))$
$= V_{.\mathbf{2}.\mathbf{2}}$.snd.fst $x\ y\ v$

▷ Prove un_sc : $(\mathbf{1} * v) = v$
$= V_{.\mathbf{2}.\mathbf{2}}$.snd.snd v

▷ Discharge $w, v, Qv, v_2, v_1, y, x, Qx, x_2, x_1, V, F$

B.7.2.2 Simple results concerning vectorspaces

▷ Introduce F; V | F-space; $x, y : F$ and $v : V$

▷ Prove ze_sc : $(\mathbf{0} * v) = \mathbf{0}$
$=$ GROUP$_4$ ((plus_sc $\mathbf{0}$ $\mathbf{0}$ v).symm.tran ((ze_plus $\mathbf{0}$).sc$_1$ v))

▷ We want to prove neg_sc : $((-x) * v) = (-(x * v))$

▷ Prove subresult Q$_1$: $(((-x) * v) + (-(-(x * v)))) = (((-x) * v) + (x * v))$
$=$ $((-x) * v)$.plus$_2$ (neg_neg $(x * v)$)

▷ Prove subresult Q$_2$: $(((-x) * v) + (x * v)) = (((-x) + x) * v)$
$=$ (plus_sc $(-x)$ x v).symm

▷ Prove subresult Q$_3$: $(((-x) + x) * v) = (\mathbf{0} * v)$
$=$ (neg_plus x).sc$_1$ v

▷ Prove subresult Q : $(((-x) * v) + (-(-(x * v)))) = \mathbf{0}$
$=$ (Q$_1$.tran Q$_2$).tran (Q$_3$.tran ze_sc)

▷ Prove as claimed neg_sc : $((-x) * v) = (-(x * v))$
$=$ (ABGROUP$_1$ $((-x) * v)$ $(-(x * v))$).snd Q

▷ Discharge but keep Q, Q$_3$, Q$_2$, Q$_1$, v, y, x

▷ Prove minus_sc : $((x - y) * v) = ((x * v) - (y * v))$
$=$ (plus_sc x $(-y)$ v).tran $((x * v)$.plus$_2$ (neg_sc y v))

▷ Discharge Q, Q$_3$, Q$_2$, Q$_1$, v, y, x, V, F

B.7.2.3 Vectorspace operators: linear combinations, independence, finite spans and dimension

▷ Introduce F; V | F-space and n | $\mathbb{N}$

▷ Allow $*$ to be written infix

▷ Define $* = (*|F|V).\mathsf{map_2le}\ n : \mathsf{map}_2\ (F\hat{}n)\ (V\hat{}n)\ (V\hat{}n)$

▷ Prove $\mathsf{lin\_com\_forms\_map_2} : ([\lambda \boldsymbol{x} : F\hat{}n]\ [\lambda \boldsymbol{v} : V\hat{}n]\ \sum\ (\boldsymbol{x} * \boldsymbol{v})).\mathsf{is\_map_2}$
$= [\lambda \boldsymbol{x}, \boldsymbol{y} \,|\, F\hat{}n]\ [\lambda H_1 : \boldsymbol{x} = \boldsymbol{y}]\ [\lambda \boldsymbol{v}, \boldsymbol{w} \,|\, V\hat{}n]\ [\lambda H_2 : \boldsymbol{v} = \boldsymbol{w}]$
$\sum.\mathsf{resp}\ (*.\mathsf{resp_2}\ H_1\ H_2)$

▷ Define $\sum^* = ([\lambda \boldsymbol{x} : F\hat{}n]\ [\lambda \boldsymbol{v} : V\hat{}n]\ \sum\ (\boldsymbol{x} * \boldsymbol{v}), \mathsf{lin\_com\_forms\_map_2} :$
$\mathsf{map}_2\ (F\hat{}n)\ (V\hat{}n)\ V) : \mathsf{map}_2\ (F\hat{}n)\ (V\hat{}n)\ V$

▷ Allow -tuple_independent_over to be written postfix

▷ Allow -span_over to be written postfix

▷ Introduce $\boldsymbol{w} : V\hat{}n$ and $L : \mathsf{subset}\ F$

▷ Define $\mathsf{is\_span\_over} = [\lambda v : V]\ \langle \exists \boldsymbol{x} : L\hat{}n \rangle\ (\sum^*\ \boldsymbol{x}\ \boldsymbol{w}) = v : V \to \mathsf{prop}$

▷ Prove span_over_forms_subset
$: \{\forall v_1, v_2 \,|\, V\}\ (v_1 = v_2) \to (\mathsf{is\_span\_over}\ v_1) \to \mathsf{is\_span\_over}\ v_2$
$= [\lambda v_1, v_2 \,|\, V]\ [\lambda Q v : v_1 = v_2]\ [\lambda V_1 : \mathsf{is\_span\_over}\ v_1]$
$(V_{1\cdot\mathbf{1}}, \mathsf{tran}\ V_{1\cdot\mathbf{2}}\ Q v : \mathsf{is\_span\_over}\ v_2)$

▷ Define -span_over $= (\mathsf{is\_span\_over}, \mathsf{span\_over\_forms\_subset} : \mathsf{subset}\ V)$
$: \mathsf{subset}\ V$

▷ Define $\mathsf{is\_independent\_over} = \{\forall \boldsymbol{x} : L\hat{}n\}\ ((\sum^*\ \boldsymbol{x}\ \boldsymbol{w}) = \mathbf{0}) \to \{\forall i\}\ (\boldsymbol{x}_\backslash i) = \mathbf{0}$
$: \mathsf{prop}$

▷ Discharge $L, \boldsymbol{w}$

▷ Introduce $L : \mathsf{subset}\ F$

▷ Prove independent_over_forms_subset
$: \mathsf{subset\_axioms}\ ([\lambda \boldsymbol{v} : V\hat{}n]\ \boldsymbol{v}.\mathsf{is\_independent\_over}\ L)$
$= [\lambda \boldsymbol{v}, \boldsymbol{w} \,|\, V\hat{}n]\ [\lambda Q : \boldsymbol{v} = \boldsymbol{w}]\ [\lambda W_1 : \boldsymbol{v}.\mathsf{is\_independent\_over}\ L]\ [\lambda \boldsymbol{x} : L\hat{}n]$
$[\lambda H : (\sum^*\ \boldsymbol{x}\ \boldsymbol{w}) = \mathbf{0}]\ W_1\ \boldsymbol{x}\ ((\sum^*.\mathsf{resps_2}\ \boldsymbol{x}\ Q).\mathsf{tran}\ H)$

▷ Define $\mathsf{independent\_over} = ([\lambda \boldsymbol{w} : V\hat{}n]\ \boldsymbol{w}.\mathsf{is\_independent\_over}\ L,$
$\mathsf{independent\_over\_forms\_subset} : \mathsf{subset}\ (V\hat{}n)) : \mathsf{subset}\ (V\hat{}n)$

▷ Discharge L, n

▷ Define spans $= [\lambda n \,|\, \mathbb{N}]\ [\lambda \boldsymbol{w} : V\hat{}n]\ [\lambda K : \mathsf{subset}\ V]\ [\lambda L : \mathsf{subset}\ F]\ K =$

$(\boldsymbol{w}\text{-}\mathsf{span\_over}\ L)$ **:** $\{\Pi n \,|\, \mathbb{N}\}\ (V\hat{}n) \to (\mathsf{subset}\ V) \to (\mathsf{subset}\ F) \to \mathsf{prop}$

▷ Define has_fin_span_over $= [\lambda K : \mathsf{subset}\ V]\ [\lambda L : \mathsf{subset}\ F]\ \langle \exists n \rangle\ \langle \exists \boldsymbol{w} : V\hat{}n \rangle$

$(\boldsymbol{w} \in (K\hat{}n)) \wedge (\boldsymbol{w}.\mathsf{spans}\ K\ L)$ **:** $(\mathsf{subset}\ V) \to (\mathsf{subset}\ F) \to \mathsf{prop}$

▷ Introduce K | subset V and L | subset F

▷ Define size_of_span $= [\lambda H : K.\mathsf{has\_fin\_span\_over}\ L]\ H_{.\mathbf{1}}$

: $(K.\mathsf{has\_fin\_span\_over}\ L) \to \mathbb{N}$

▷ Define spanning_tuple

$= [\lambda H : K.\mathsf{has\_fin\_span\_over}\ L]\ (H_{.\mathbf{2}\cdot\mathbf{1}}, H_{.\mathbf{2}\cdot\mathbf{2}}.\mathsf{fst} : K\hat{}(\mathsf{size\_of\_span}\ H))$

: $\{\Pi H : K.\mathsf{has\_fin\_span\_over}\ L\}\ K\hat{}(\mathsf{size\_of\_span}\ H)$

▷ Discharge L, K, V

▷ Define -tuple_independent_over $= [\lambda n]\ [\lambda L : \mathsf{subset}\ F]\ [\lambda V]$

independent_over$|V|n$ L **:** $\{\Pi n\}\ (\mathsf{subset}\ F) \to \{\Pi V\}\ \mathsf{subset}\ (V\hat{}n)$

▷ Introduce V | F-space and W **:** subset V

▷ Define has_basis_over $= [\lambda L : \mathsf{subset}\ F]\ [\lambda n \,|\, \mathbb{N}]\ [\lambda \boldsymbol{w} : V\hat{}n]$

$\bigwedge_3\ (\boldsymbol{w} \in (W\hat{}n))\ (\boldsymbol{w}.\mathsf{spans}\ W\ L)\ (\boldsymbol{w} \in (n\text{-}\mathsf{tuple\_independent\_over}\ L\ V))$

: $(\mathsf{subset}\ F) \to \{\Pi n \,|\, \mathbb{N}\}\ (V\hat{}n) \to \mathsf{prop}$

▷ Define has_fin_dim_over $= [\lambda L : \mathsf{subset}\ F]\ [\lambda n]\ \langle \exists \boldsymbol{w} : V\hat{}n \rangle\ \mathsf{has\_basis\_over}\ L\ \boldsymbol{w}$

: $(\mathsf{subset}\ F) \to \mathbb{N} \to \mathsf{prop}$

▷ Define is_fin_dim_over $= [\lambda L : \mathsf{subset}\ F]\ \langle \exists n \rangle\ \mathsf{has\_fin\_dim\_over}\ L\ n$

: $(\mathsf{subset}\ F) \to \mathsf{prop}$

▷ Discharge W

▷ Introduce W | subset V and L | subset F

▷ Define dim $= [\lambda H : W.\mathsf{is\_fin\_dim\_over}\ L]\ H_{.\mathbf{1}}$ **:** $(W.\mathsf{is\_fin\_dim\_over}\ L) \to \mathbb{N}$

▷ Define basis $= [\lambda H : W.\mathsf{is\_fin\_dim\_over}\ L]\ (H_{.\mathbf{2}\cdot\mathbf{1}}, H_{.\mathbf{2}\cdot\mathbf{2}}.\mathsf{fst}_3 : W\hat{}(\mathsf{dim}\ H))$

: $\{\Pi H : W.\mathsf{is\_fin\_dim\_over}\ L\}\ W\hat{}(\mathsf{dim}\ H)$

▷ Suppose H **:** $W.\mathsf{is\_fin\_dim\_over}\ L$

▷ Prove fin_dim : W.has_fin_dim_over L (dim H)
= $H._{\mathbf{2}}$

▷ Prove has_basis : W.has_basis_over L H.basis
= $H._{\mathbf{2}\cdot\mathbf{2}}$

▷ Prove spanning : H.basis.spans W L
= has_basis.snd$_3$

▷ Prove span_fin_dim : W.has_fin_span_over L
= (dim H, H.basis, pair has_basis.fst$_3$ spanning : W.has_fin_span_over L)

▷ Prove independence : H.basis $\in$ ((dim H)-tuple_independent_over L V)
= has_basis.thd$_3$

▷ Discharge H, L, W, V

▷ Introduce K, L | subset F

▷ Define coercion tuple_self_space = $[\lambda n \,|\, \mathbb{N}]\ [\lambda \boldsymbol{x} : F\hat{}n]\ \boldsymbol{x}$
: $\{\Pi n \,|\, \mathbb{N}\}\ (F\hat{}n) \to F$.as_space_over_itself$\hat{}n$

▷ Discharge L, K, F

B.7.2.4 Results concerning vectorspace operators

▷ Introduce F and V | F-space

▷ Introduce n | $\mathbb{N}$; $\boldsymbol{w}$: $V\hat{}n$; K, L | subset F and suppose H : $K \subseteq L$

▷ Introduce v : $\boldsymbol{w}$-span_over K and let $\boldsymbol{x}$ = POWER$_1$s H n $(v.\mathsf{ev})._{\mathbf{1}}$
: $(v.\mathsf{ev})._{\mathbf{1}} \in (L\hat{}n)$

▷ Prove subresult P : $(\sum^* \boldsymbol{x}\ \boldsymbol{w}) = v$
= $(v.\mathsf{ev})._{\mathbf{2}}$

▷ Prove SPAN$_2$s : $v \in (\boldsymbol{w}$-span_over $L)$
= ($\boldsymbol{x}$, P : $v \in (\boldsymbol{w}$-span_over $L)$)

▷ Discharge P, $\boldsymbol{x}, v$

▷ Prove INDEP$_1$s
: (n-tuple_independent_over L V) $\subseteq$ (n-tuple_independent_over K V)
= $[\lambda \boldsymbol{v} : n$-tuple_independent_over L $V]$ $[\lambda \boldsymbol{x} : K\hat{}n]$ $[\lambda Q : (\sum^* \boldsymbol{x}\ \boldsymbol{v}) = \mathbf{0}]$
$\boldsymbol{v}$.ev (POWER$_1$s H n $\boldsymbol{x}$) Q

▷ Discharge but keep H, L, K

▷ Prove SPAN_2 : $(K = L) \to (\boldsymbol{w}\text{-span\_over } K) = (\boldsymbol{w}\text{-span\_over } L)$
= $[\lambda Q : K = L]$ pair ($\mathsf{SPAN}_2\mathsf{s}\ Q.\mathsf{fst}$) ($\mathsf{SPAN}_2\mathsf{s}\ Q.\mathsf{snd}$)

▷ Prove INDEP_1 : $(K = L) \to$
$(n\text{-tuple\_independent\_over } K\ V) = (n\text{-tuple\_independent\_over } L\ V)$
= $[\lambda Q : K = L]$ pair ($\mathsf{INDEP}_1\mathsf{s}\ Q.\mathsf{snd}$) ($\mathsf{INDEP}_1\mathsf{s}\ Q.\mathsf{fst}$)

▷ Discharge $H, L, K, \boldsymbol{w}, n$

▷ Introduce J, K | subset V; suppose Q : $J = K$ and introduce L : subset F

▷ Introduce n | $\mathbb{N}$ and $\boldsymbol{w}$ | $V\text{^}n$

▷ Suppose H : J.has_basis_over $L\ \boldsymbol{w}$

▷ We want to prove $\mathsf{FINDIM}_1\mathsf{b}$: K.has_basis_over $L\ \boldsymbol{w}$

▷ Prove subresult $\mathsf{FINDIM}_1\mathsf{b}_1$: $\boldsymbol{w} \in (K\text{^}n)$
= $\mathsf{POWER}_1\mathsf{s}\ Q.\mathsf{fst}\ n\ H.\mathsf{fst}_3$

▷ Prove subresult $\mathsf{FINDIM}_1\mathsf{b}_2$: $K = (\boldsymbol{w}\text{-span\_over } L)$
= Q.equal_subs_symm.equal_subs_tran $H.\mathsf{snd}_3$

▷ Prove subresult $\mathsf{FINDIM}_1\mathsf{b}_3$: $\boldsymbol{w} \in (n\text{-tuple\_independent\_over } L\ V)$
= $H.\mathsf{thd}_3$

▷ Prove as claimed $\mathsf{FINDIM}_1\mathsf{b}$: K.has_basis_over $L\ \boldsymbol{w}$
= $\mathsf{pair}_3\ \mathsf{FINDIM}_1\mathsf{b}_1\ \mathsf{FINDIM}_1\mathsf{b}_2\ \mathsf{FINDIM}_1\mathsf{b}_3$

▷ Discharge $\mathsf{FINDIM}_1\mathsf{b}_3, \mathsf{FINDIM}_1\mathsf{b}_2, \mathsf{FINDIM}_1\mathsf{b}_1, H, \boldsymbol{w}$

▷ Prove $\mathsf{FINDIM}_1\mathsf{a}$: (J.has_fin_dim_over $L\ n$) $\to$ K.has_fin_dim_over $L\ n$
= $[\lambda H : J.\mathsf{has\_fin\_dim\_over}\ L\ n]$ ($H.\mathbf{1}, \mathsf{FINDIM}_1\mathsf{b}\ H.\mathbf{2} : K.\mathsf{has\_fin\_dim\_over}\ L\ n$)

▷ Discharge n

▷ Prove FINDIM_1 : (J.is_fin_dim_over L) $\to$ K.is_fin_dim_over L
= $[\lambda H : J.\mathsf{is\_fin\_dim\_over}\ L]$ (dim $H, \mathsf{FINDIM}_1\mathsf{a}\ H.\mathbf{2} : K.\mathsf{is\_fin\_dim\_over}\ L$)

▷ Discharge L, Q, K, J

▷ Introduce J : subset V; K, L | subset F and suppose Q : $K = L$

▷ Introduce n | $\mathbb{N}$ and $\boldsymbol{w}$ | $V\text{^}n$

▷ Suppose H : J.has_basis_over $K\ \boldsymbol{w}$

▷ We want to prove $\mathsf{FINDIM}_2\mathsf{b}$: J.has_basis_over $L\ \boldsymbol{w}$

▷ Prove subresult $\mathsf{FINDIM}_2\mathsf{b}_1$: $\boldsymbol{w} \in (J\text{^}n)$
= $H.\mathsf{fst}_3$

▷ Prove subresult $\mathsf{FINDIM_2b_2} : J = (\boldsymbol{w}\text{-}\mathsf{span\_over}\ L)$
$= H.\mathsf{snd_3.equal\_subs\_tran}\ (\boldsymbol{w}.\mathsf{SPAN_2}\ Q)$

▷ Prove subresult $\mathsf{FINDIM_2b_3} : \boldsymbol{w} \in (n\text{-}\mathsf{tuple\_independent\_over}\ L\ V)$
$= \mathsf{INDEP_1s}\ Q.\mathsf{snd}\ (\mathsf{make}\ H.\mathsf{thd_3})$

▷ Prove as claimed $\mathsf{FINDIM_2b} : J.\mathsf{has\_basis\_over}\ L\ \boldsymbol{w}$
$= \mathsf{pair_3}\ \mathsf{FINDIM_2b_1}\ \mathsf{FINDIM_2b_2}\ \mathsf{FINDIM_2b_3}$

▷ Discharge $\mathsf{FINDIM_2b_3}, \mathsf{FINDIM_2b_2}, \mathsf{FINDIM_2b_1}, H, \boldsymbol{w}$

▷ Prove $\mathsf{FINDIM_2a} : (J.\mathsf{has\_fin\_dim\_over}\ K\ n) \to J.\mathsf{has\_fin\_dim\_over}\ L\ n$
$= [\lambda H : J.\mathsf{has\_fin\_dim\_over}\ K\ n]\ (H._{\mathbf{1}}, \mathsf{FINDIM_2b}\ H._{\mathbf{2}} : J.\mathsf{has\_fin\_dim\_over}\ L\ n)$

▷ Discharge n

▷ Prove $\mathsf{FINDIM_2} : (J.\mathsf{is\_fin\_dim\_over}\ K) \to J.\mathsf{is\_fin\_dim\_over}\ L$
$= [\lambda H : J.\mathsf{is\_fin\_dim\_over}\ K]\ (\mathsf{dim}\ H, \mathsf{FINDIM_2a}\ H._{\mathbf{2}} : J.\mathsf{is\_fin\_dim\_over}\ L)$

▷ Discharge Q, L, K, J, V, F

B.7.2.5 The character tuple δ

▷ Introduce F; $n \mid \mathbb{N}$ and i

▷ Define $\mathsf{delta\_fun} = [\lambda j]\ \mathsf{if}\ (\mathsf{n\_discrete}\ n\ i\ j)\ (\mathbf{1}|F)\ (\mathbf{0}|F) : n \to F._{\mathbf{1}.\mathbf{1}.\mathbf{1}}$

▷ Prove subresult $\mathsf{delta\_fun\_true} : \{\Pi j\}\ (i = j) \to (\mathsf{delta\_fun}\ j) = \mathbf{1}$
$= [\lambda j]\ [\lambda Q : i = j]\ \mathsf{IF_1}\ (\mathbf{1}|F)\ \mathbf{0}\ (\mathsf{n\_discrete}\ n\ i\ j)\ Q$

▷ Prove subresult $\mathsf{delta\_fun\_false} : \{\Pi j\}\ (i \neq j) \to (\mathsf{delta\_fun}\ j) = \mathbf{0}$
$= [\lambda j]\ [\lambda Q : i \neq j]\ \mathsf{IF_0}\ (\mathbf{1}|F)\ \mathbf{0}\ (\mathsf{n\_discrete}\ n\ i\ j)\ Q$

▷ Prove $\mathsf{delta\_forms\_tuple} : \mathsf{delta\_fun}.(\mathsf{is\_map}|n)$
$= [\lambda j_1, j_2 : n]\ [\lambda Q : j_1 = j_2]\ \mathsf{case}\ (\mathsf{n\_discrete}\ n\ i\ j_1)$
$([\lambda Q' : i = j_1]\ (\mathsf{delta\_fun\_true}\ j_1\ Q').\mathsf{tran}\ (\mathsf{delta\_fun\_true}\ j_2\ (Q'.\mathsf{tran}\ Q)).\mathsf{symm})$
$([\lambda Q' : i \neq j_1]\ (\mathsf{delta\_fun\_false}\ j_1\ Q').\mathsf{tran}$
$(\mathsf{delta\_fun\_false}\ j_2\ ([\lambda H : i = j_2]\ Q'\ (H.\mathsf{tran}\ Q.\mathsf{symm}))).\mathsf{symm})$

▷ Define $\mathsf{delta\_tuple} = (\mathsf{delta\_fun}, \mathsf{delta\_forms\_tuple} : F^{\wedge}n) : F^{\wedge}n$

▷ Discharge i

▷ Prove $\mathsf{delta\_forms\_map} : \mathsf{delta\_tuple}.(\mathsf{is\_map}|n)$
$= [\lambda i_1, i_2 \,|\, n]\ [\lambda Q : i_1 = i_2]\ [\lambda j]\ \mathsf{case}\ (\mathsf{n\_discrete}\ n\ i_1\ j)\ ([\lambda Q' : i_1 = j]$
$(\mathsf{delta\_fun\_true}\ i_1\ j\ Q').\mathsf{tran}\ (\mathsf{delta\_fun\_true}\ i_2\ j\ (Q.\mathsf{symm.tran}\ Q')).\mathsf{symm})$
$([\lambda Q' : i_1 \neq j]\ (\mathsf{delta\_fun\_false}\ i_1\ j\ Q').\mathsf{tran}$
$(\mathsf{delta\_fun\_false}\ i_2\ j\ ([\lambda H : i_2 = j]\ Q'\ (Q.\mathsf{tran}\ H))).\mathsf{symm})$

▷ Define $\boldsymbol{\delta} = (\mathsf{delta\_tuple}, \mathsf{delta\_forms\_map} : \mathsf{map}\ n\ (F^{\wedge}n)) : \mathsf{map}\ n\ (F^{\wedge}n)$

▷ Prove DELTA_0 **:** $\{\forall i, j \,|\, n\}\ (i \neq j) \to ((\boldsymbol{\delta}\ i)_\backslash j) = \mathbf{0}$
$= [\lambda i, j \,|\, n]$ delta_fun_false $i\ j$

▷ Prove DELTA_1 **:** $\{\forall i, j \,|\, n\}\ (i = j) \to ((\boldsymbol{\delta}\ i)_\backslash j) = \mathbf{1}$
$= [\lambda i, j \,|\, n]$ delta_fun_true $i\ j$

▷ Discharge but keep delta_fun_false, delta_fun_true, n

▷ Prove delta_tail **:** $\{\forall i\}\ (\boldsymbol{\delta}\ i{+}\underline{1}).\mathsf{tl} = (\boldsymbol{\delta}\ i)$
$= [\lambda i, j]$ $\mathsf{IF}_2\ (\mathbf{1}|F)\ \mathbf{0}$ (n_discrete $n{+}1\ i{+}\underline{1}\ j{+}\underline{1}$) (n_discrete $n\ i\ j$)

▷ Discharge delta_fun_false, delta_fun_true, n, F

B.7.2.6 Results concerning the character tuple δ

▷ Introduce F and V | F-space

▷ Let $\mathsf{p} = [\lambda n]\ \{\forall \boldsymbol{v} : V^\wedge n\}\ \{\forall i\}\ (\sum\ ((\boldsymbol{\delta}\ i) * \boldsymbol{v})) = (\boldsymbol{v}_\backslash i) : \mathbb{N} \to$ prop

▷ Prove subresult P_0 **:** p 0
$= [\lambda \boldsymbol{v} : V^\wedge 0]\ [\lambda i : 0]$ ex_O $((\sum\ ((\boldsymbol{\delta}\ i) * \boldsymbol{v})) = (\boldsymbol{v}_\backslash i))\ i$

▷ Introduce n; suppose ih **:** p n; introduce $\boldsymbol{v}$ **:** $V^\wedge n{+}1$ and i **:** $n{+}1$

▷ Suppose Q **:** equal_in $\mathbb{N}\ (\underline{0}\ n)\ i$

▷ Prove subresult Q_1 **:** $((\boldsymbol{v}_\backslash i) + \mathbf{0}) = (\boldsymbol{v}_\backslash i)$
$= (\boldsymbol{v}_\backslash i).$plus_ze

▷ Prove subresult Q_2 **:** $(\mathbf{1} * (\boldsymbol{v}_\backslash i)) = (\boldsymbol{v}_\backslash i)$
$=$ un_sc $(\boldsymbol{v}_\backslash i)$

▷ Prove subresult Q_3 **:** $((\boldsymbol{\delta}|F\ i)_\backslash(\underline{0}\ n)) = \mathbf{1}$
$= \mathsf{DELTA}_1|F\ Q.$symm

▷ Prove subresult Q_4 **:** $(\boldsymbol{v}\ (\underline{0}\ n)) = (\boldsymbol{v}\ i)$
$= \boldsymbol{v}.$resp Q

▷ Prove subresult Q_5 **:** $(((\boldsymbol{\delta}|F\ i)_\backslash(\underline{0}\ n)) * (\boldsymbol{v}\ (\underline{0}\ n))) = (\mathbf{1} * (\boldsymbol{v}\ i))$
$=$ sc_resp $\mathsf{Q}_3\ \mathsf{Q}_4$

▷ Prove subresult Q_6 **:** $(\sum|V\ (\mathbf{0}^\wedge n)) = \mathbf{0}$
$= \mathsf{LINSUM}_0|V\ n$

▷ Introduce j

▷ Prove subresult Q_8 **:** $(\mathbf{0} * (\boldsymbol{v}_\backslash j{+}\underline{1})) = \mathbf{0}$
$=$ ze_sc $(\boldsymbol{v}_\backslash j{+}\underline{1})$

▷ Prove subresult Q_9 : $((\boldsymbol{\delta}|F\ i)_\backslash j{+}\underline{1}) = \mathbf{0}$
= $\mathsf{DELTA}_0|F\ ([\lambda Q' : i = j{+}\underline{1}]\ Q.\mathsf{tran}\ Q')$

▷ Prove subresult Q_{10} : $(((\boldsymbol{\delta}|F\ i)_\backslash j{+}\underline{1}) * (\boldsymbol{v}_\backslash j{+}\underline{1})) = (\mathbf{0} * (\boldsymbol{v}_\backslash j{+}\underline{1}))$
= $\mathsf{Q}_9.\mathsf{sc}_1\ (\boldsymbol{v}_\backslash j{+}\underline{1})$

▷ Discharge j

▷ Prove subresult Q_7 : $((\boldsymbol{\delta}\ i) * \boldsymbol{v}).\mathsf{tl} = (\mathbf{0}^\wedge n)$
= $[\lambda j]\ (\mathsf{Q}_{10}\ j).\mathsf{tran}\ (\mathsf{Q}_8\ j)$

▷ Prove subresult Q_{11} : $(\sum ((\boldsymbol{\delta}\ i) * \boldsymbol{v}).\mathsf{tl}) = \mathbf{0}$
= $(\sum.\mathsf{resp}\ \mathsf{Q}_7).\mathsf{tran}\ \mathsf{Q}_6$

▷ Prove subresult C_0
: $((((\boldsymbol{\delta}|F\ i)_\backslash(\underline{0}\ n)) * (\boldsymbol{v}\ (\underline{0}\ n))) + (\sum ((\boldsymbol{\delta}\ i) * \boldsymbol{v}).\mathsf{tl})) = (\boldsymbol{v}_\backslash i)$
= $(\mathsf{plus\_resp}\ (\mathsf{Q}_5.\mathsf{tran}\ \mathsf{Q}_2)\ \mathsf{Q}_{11}).\mathsf{tran}\ \mathsf{Q}_1$

▷ Discharge Q

▷ Suppose H : $\langle\exists k\rangle$ $\mathsf{equal\_in}\ n{+}1\ k{+}\underline{1}\ i$ and let $\mathsf{k} = H_{.\mathbf{1}} : n$

▷ Prove subresult Q : $\mathsf{k}{+}\underline{1} = i$
= $H_{.\mathbf{2}}$

▷ Prove subresult Q_{12} : $(\sum ((\boldsymbol{\delta}\ \mathsf{k}) * \boldsymbol{v}.\mathsf{tl})) = (\boldsymbol{v}.\mathsf{tl}_\backslash\mathsf{k})$
= $ih\ \boldsymbol{v}.\mathsf{tl}\ \mathsf{k}$

▷ Prove subresult Q_{13} : $(\boldsymbol{\delta}|F\ i) = (\boldsymbol{\delta}|F\ \mathsf{k}{+}\underline{1})$
= $(\boldsymbol{\delta}|F).\mathsf{resp}\ \mathsf{Q}.\mathsf{symm}$

▷ Prove subresult Q_{14} : $(\boldsymbol{\delta}\ \mathsf{k}{+}\underline{1}).\mathsf{tl} = (\boldsymbol{\delta}\ \mathsf{k})$
= $\mathsf{delta\_tail}|F\ \mathsf{k}$

▷ Prove subresult Q_{15} : $(\mathsf{tl}\ (\boldsymbol{\delta}|F\ i)) = (\mathsf{tl}\ (\boldsymbol{\delta}|F\ \mathsf{k}{+}\underline{1}))$
= $\mathsf{tl}.\mathsf{resp}\ \mathsf{Q}_{13}$

▷ Prove subresult Q_{16} : $(\sum ((\boldsymbol{\delta}\ i).\mathsf{tl} * \boldsymbol{v}.\mathsf{tl})) = (\sum ((\boldsymbol{\delta}\ \mathsf{k}) * \boldsymbol{v}.\mathsf{tl}))$
= $\sum.\mathsf{resp}\ ((\mathsf{Q}_{15}.\mathsf{tran}\ \mathsf{Q}_{14}).(*.\mathsf{resps}_1)\ \boldsymbol{v}.\mathsf{tl})$

▷ Prove subresult Q_{18} : $(\sum ((\boldsymbol{\delta}\ i) * \boldsymbol{v}).\mathsf{tl}) = (\sum ((\boldsymbol{\delta}\ i).\mathsf{tl} * \boldsymbol{v}.\mathsf{tl}))$
= $\sum.\mathsf{resp}\ (\mathsf{map}_2\mathsf{\_tail}\ (\boldsymbol{\delta}\ i)\ \boldsymbol{v}\ *).\mathsf{symm}$

▷ Prove subresult Q_{19} : $(\boldsymbol{v}.\mathsf{tl}_\backslash\mathsf{k}) = (\boldsymbol{v}_\backslash i)$
= $\boldsymbol{v}.\mathsf{resp}\ \mathsf{Q}$

▷ Prove subresult Q_{20} : $(\sum ((\boldsymbol{\delta}\ i) * \boldsymbol{v}).\mathsf{tl}) = (\boldsymbol{v}_\backslash i)$
= $(\mathsf{Q}_{18}.\mathsf{tran}\ \mathsf{Q}_{16}).\mathsf{tran}\ (\mathsf{Q}_{12}.\mathsf{tran}\ \mathsf{Q}_{19})$

▷ Prove subresult Q_{21} : $(\boldsymbol{\delta}|F\ i).\mathsf{hd} = \mathbf{0}$
= $\mathsf{DELTA}_0|F\ ([\lambda Q' : i = (\underline{0}\ n)]\ \mathsf{Q.tran}\ Q')$

▷ Prove subresult Q_{22} : $((\boldsymbol{\delta}|F\ i).\mathsf{hd} * \boldsymbol{v}.\mathsf{hd}) = (\mathbf{0} * \boldsymbol{v}.\mathsf{hd})$
= $\mathsf{Q}_{21}.\mathsf{sc}_1\ \boldsymbol{v}.\mathsf{hd}$

▷ Prove subresult Q_{23} : $(\mathbf{0} * \boldsymbol{v}.\mathsf{hd}) = \mathbf{0}$
= ze_sc $\boldsymbol{v}.\mathsf{hd}$

▷ Prove subresult Q_{24} : $(\sum ((\boldsymbol{\delta}\ i) * \boldsymbol{v})) = (\mathbf{0} + (\boldsymbol{v}_\backslash i))$
= plus_resp (Q_{22}.tran Q_{23}) Q_{20}

▷ Prove subresult Q_{25} : $(\mathbf{0} + (\boldsymbol{v}_\backslash i)) = (\boldsymbol{v}_\backslash i)$
= ze_plus $(\boldsymbol{v}_\backslash i)$

▷ Prove subresult Cs : $(\sum ((\boldsymbol{\delta}\ i) * \boldsymbol{v})) = (\boldsymbol{v}_\backslash i)$
= Q_{24}.tran Q_{25}

▷ Discharge H

▷ Prove subresult Ps
: $((((\boldsymbol{\delta}|F\ i)_\backslash(\underline{0}\ n)) * (\boldsymbol{v}\ (\underline{0}\ n))) + (\sum ((\boldsymbol{\delta}\ i) * \boldsymbol{v}).\mathsf{tl})) = (\boldsymbol{v}_\backslash i)$
= case ($\mathsf{CANON}_2\ i$) C_0 Cs

▷ Discharge $i, \boldsymbol{v}, ih, n$

▷ Prove DELTA_2 : $\{\forall n \,|\, \mathbb{N}\}\ \{\forall \boldsymbol{v} : V^\wedge n\}\ \{\forall i\}\ (\sum ((\boldsymbol{\delta}\ i) * \boldsymbol{v})) = (\boldsymbol{v}_\backslash i)$
= N_elim p P_0 Ps

▷ Discharge $\mathsf{Ps}, \mathsf{Cs}, \mathsf{Q}_{25}, \mathsf{Q}_{24}, \mathsf{Q}_{23}, \mathsf{Q}_{22}, \mathsf{Q}_{21}, \mathsf{Q}_{20}, \mathsf{Q}_{19}, \mathsf{Q}_{18}, \mathsf{Q}_{16}, \mathsf{Q}_{15}, \mathsf{Q}_{14}, \mathsf{Q}_{13}, \mathsf{Q}_{12}, \mathsf{Q}, \mathsf{k}, \mathsf{C}_0, \mathsf{Q}_{11}, \mathsf{Q}_7, \mathsf{Q}_{10}, \mathsf{Q}_9, \mathsf{Q}_8, \mathsf{Q}_6, \mathsf{Q}_5, \mathsf{Q}_4, \mathsf{Q}_3, \mathsf{Q}_2, \mathsf{Q}_1, \mathsf{P}_0, \mathsf{p}$

▷ Introduce L and n | $\mathbb{N}$

▷ Prove DELTA_3 : $\{\forall i\}\ (\boldsymbol{\delta}\ i) \in (L^\wedge n)$
= $[\lambda i, j]\ [\delta\mathsf{D} = \mathsf{n\_discrete}\ n\ i\ j]$
case D ($[\lambda H : i = j]$ eq_closed$|F|L$ ($\mathsf{DELTA}_1\ H$).symm un_closed)
($[\lambda H : i \neq j]$ eq_closed$|F|L$ ($\mathsf{DELTA}_0\ H$).symm ze_closed)

▷ Discharge n, L, V, F

B.7.2.7 Results concerning independence

▷ Introduce F; V | F-space; K | subset V; L | subfield F and n | $\mathbb{N}$

▷ Allow $-$ to be written infix

▷ Define $- =$ $[\lambda G \,|\, \mathsf{abelian\_group}]\ (-|G).\mathsf{map}_2\mathsf{le}\ n$
: $\{\Pi G \,|\, \mathsf{abelian\_group}\}\ \mathsf{map}_2\ (G^\wedge n)\ (G^\wedge n)\ (G^\wedge n)$

▷ Introduce $\boldsymbol{v}$: n-tuple_independent_over L V; i, j | n and suppose Q : $(\boldsymbol{v}\_i) = (\boldsymbol{v}\_j)$

▷ Let $\mathsf{di} = \boldsymbol{\delta}|F\ i : F_{\cdot \mathbf{1} \cdot \mathbf{1} \cdot \mathbf{1}}{}^{\wedge} n$ and $\mathsf{dj} = \boldsymbol{\delta}|F\ j : F_{\cdot \mathbf{1} \cdot \mathbf{1} \cdot \mathbf{1}}{}^{\wedge} n$

▷ Let $\mathsf{dvi} = \sum^{*} \mathsf{di}\ \boldsymbol{v} : V_{\cdot \mathbf{1} \cdot \mathbf{1} \cdot \mathbf{1}}$ and $\mathsf{dvj} = \sum^{*} \mathsf{dj}\ \boldsymbol{v} : V_{\cdot \mathbf{1} \cdot \mathbf{1} \cdot \mathbf{1}}$

▷ Prove subresult $\mathsf{P}_1\mathsf{i} : \mathsf{dvi} = (\boldsymbol{v}\_i)$
= DELTA$_2$ $\boldsymbol{v}$ i

▷ Prove subresult $\mathsf{P}_1\mathsf{j} : \mathsf{dvj} = (\boldsymbol{v}\_j)$
= DELTA$_2$ $\boldsymbol{v}$ j

▷ Prove subresult $\mathsf{P}_1 : \mathsf{dvi} = \mathsf{dvj}$
= (P_1i.tran Q).tran P_1j.symm

▷ Prove subresult $\mathsf{P}_2 : (\mathsf{dvi} - \mathsf{dvj}) = \mathbf{0}$
= (ABGROUP$_1$ dvi dvj).fst P_1

▷ Prove subresult $\mathsf{P}_3 : (\mathsf{dvi} - \mathsf{dvj}) = (\sum ((\mathsf{di} * \boldsymbol{v}) - (\mathsf{dj} * \boldsymbol{v})))$
= LINSUM$_4$ − ABGROUP$_3$ (di $*$ $\boldsymbol{v}$) (dj $*$ $\boldsymbol{v}$)

▷ Prove subresult $\mathsf{P}_4 : ((\mathsf{di} - \mathsf{dj}) * \boldsymbol{v}) = ((\mathsf{di} * \boldsymbol{v}) - (\mathsf{dj} * \boldsymbol{v}))$
= [λk] minus_sc (di_k) (dj_k) ($\boldsymbol{v}\_k$)

▷ Prove subresult $\mathsf{P}_0 : (\sum^{*} (\mathsf{di} - \mathsf{dj})\ \boldsymbol{v}) = \mathbf{0}$
= ($\sum$.resp P_4).tran (P_3.symm.tran P_2)

▷ Introduce k

▷ Prove subresult $\mathsf{P}_5\mathsf{i} : (\mathsf{di}\_k) \in L$
= DELTA$_3$ L i k

▷ Prove subresult $\mathsf{P}_5\mathsf{j} : (\mathsf{dj}\_k) \in L$
= DELTA$_3$ L j k

▷ Discharge k

▷ Prove subresult $\mathsf{P}_5 : (\mathsf{di} - \mathsf{dj}) \in (L^{\wedge} n)$
= [λk] minus_closed (P_5i k) (P_5j k)

▷ Prove subresult $\mathsf{P}_6 : \{\forall k\}\ ((\mathsf{di} - \mathsf{dj})\_k) = \mathbf{0}$
= $\boldsymbol{v}$.ev P_5 P_0

▷ Suppose $H : i \neq j$

▷ Prove subresult $\mathsf{P}_7\mathsf{i} : (\mathsf{di}\_j) = \mathbf{0}$
= IF$_0$ ($\mathbf{1}|F$) $\mathbf{0}$ (n_discrete n i j) H

▷ Prove subresult $\mathsf{P}_7\mathsf{j} : (\mathsf{dj}\_j) = \mathbf{1}$
= IF$_1$ ($\mathbf{1}|F$) $\mathbf{0}$ (n_discrete n j j) (refl j)

▷ Prove subresult $\mathsf{P}_7 : (\mathsf{di}\backslash j) = (\mathsf{dj}\backslash j)$
$= (\mathsf{ABGROUP}_1\ (\mathsf{di}\backslash j)\ (\mathsf{dj}\backslash j)).\mathsf{snd}\ (\mathsf{P}_6\ j)$

▷ Prove subresult $\mathsf{P}_8 : \mathbf{0} = \mathbf{1}$
$= \mathsf{P}_7\mathsf{i.symm.tran}\ (\mathsf{P}_7.\mathsf{tran}\ \mathsf{P}_7\mathsf{j})$

▷ Prove subresult $\mathsf{P}_9 : i = j$
$= \mathsf{ex\_falso}\ (i = j)\ (\mathsf{nontriv}\ \mathsf{P}_8.\mathsf{symm})$

▷ Discharge H

▷ Prove $\mathsf{INDEP}_2 : i = j$
$= \mathsf{case}\ (\mathsf{n\_discrete}\ n\ i\ j)\ ([\lambda H : i = j]\ H)\ \mathsf{P}_9$

▷ Discharge $\mathsf{P}_9, \mathsf{P}_8, \mathsf{P}_7, \mathsf{P}_7\mathsf{j}, \mathsf{P}_7\mathsf{i}, \mathsf{P}_6, \mathsf{P}_5, \mathsf{P}_5\mathsf{j}, \mathsf{P}_5\mathsf{i}, \mathsf{P}_0, \mathsf{P}_4, \mathsf{P}_3, \mathsf{P}_2, \mathsf{P}_1, \mathsf{P}_1\mathsf{j}, \mathsf{P}_1\mathsf{i}, \mathsf{dvj}, \mathsf{dvi}, \mathsf{dj}, \mathsf{di}, Q, j, i$

▷ We want to prove $\mathsf{INDEP}_3\mathsf{a} : \boldsymbol{v} \cong n$

▷ Let $\mathsf{phi\_fun} = [\lambda i]\ (\boldsymbol{v}\backslash i).\mathsf{as\_el\_of}\ \boldsymbol{v}.\mathsf{rep}\ (i, \mathsf{refl}\ (\boldsymbol{v}\backslash i) : (\boldsymbol{v}\backslash i) \in \boldsymbol{v}) : n \to \boldsymbol{v}.\mathsf{rep}$

▷ Prove subresult $\mathsf{phi\_forms\_map}$
$: \{\forall x_1, x_2 \,|\, n\}\ (x_1 = x_2) \to (\boldsymbol{v}.\mathsf{rep}\ x_1) = (\boldsymbol{v}.\mathsf{rep}\ x_2)$
$= \boldsymbol{v}.\mathsf{rep.resp}$

▷ Let $\phi = (\mathsf{phi\_fun}, \mathsf{phi\_forms\_map} : \mathsf{map}\ n\ \boldsymbol{v}.\mathsf{rep}) : \mathsf{map}\ n\ \boldsymbol{v}.\mathsf{rep}$

▷ Let $\mathsf{phi'\_fun} = [\lambda v : \boldsymbol{v}.\mathsf{rep}]\ (v.\mathsf{ev})._{\mathbf{1}} : \boldsymbol{v}.\mathsf{rep} \to n$

▷ We want to prove subresult $\mathsf{phi'\_forms\_map} : \mathsf{phi'\_fun}.(\mathsf{is\_map}|\boldsymbol{v}.\mathsf{rep}|n)$

▷ Introduce $v_1, v_2 \mid \boldsymbol{v}.\mathsf{rep}$ and $Q : v_1 = v_2$

▷ Prove subresult $\mathsf{P}_1 : (\boldsymbol{v}\backslash(\mathsf{phi'\_fun}\ v_1)) = (\boldsymbol{v}\backslash(\mathsf{phi'\_fun}\ v_2))$
$= ((v_1.\mathsf{ev})._{\mathbf{2}}.\mathsf{tran}\ Q).\mathsf{tran}\ (v_2.\mathsf{ev})._{\mathbf{2}}.\mathsf{symm}$

▷ Discharge to prove subresult as claimed $\mathsf{phi'\_forms\_map}$
$: \mathsf{phi'\_fun}.(\mathsf{is\_map}|\boldsymbol{v}.\mathsf{rep}|n)$
using
$\mathsf{INDEP}_2\ \mathsf{P}_1 : (\mathsf{phi'\_fun}\ v_1) = (\mathsf{phi'\_fun}\ v_2)$

▷ Discharge Q, v_2, v_1

▷ Let $\phi' = (\mathsf{phi'\_fun}, \mathsf{phi'\_forms\_map} : \mathsf{map}\ \boldsymbol{v}.\mathsf{rep}\ n) : \mathsf{map}\ \boldsymbol{v}.\mathsf{rep}\ n$

▷ Prove subresult $\mathsf{P}_{2\text{-}1} : (\phi' \circ \phi) = \mathsf{identity}$
$= [\lambda i]\ \mathsf{refl}\ i$

▷ Prove subresult $\mathsf{P}_{2\text{-}2} : (\phi \circ \phi') = \mathsf{identity}$
$= [\lambda v : \boldsymbol{v}.\mathsf{rep}]\ (v.\mathsf{ev})._{\mathbf{2}}$

▷ Prove subresult P_2 : $\phi \in (\mathsf{iso}\ n\ \boldsymbol{v}.\mathsf{rep})$
$= (\phi', \mathsf{pair}\ \mathsf{P}_{2\text{-}1}\ \mathsf{P}_{2\text{-}2} : \phi \in (\mathsf{iso}\ n\ \boldsymbol{v}.\mathsf{rep}))$

▷ Let $\mathsf{phi\_iso} = \mathsf{make}\ \mathsf{P}_2$: $\mathsf{iso}\ n\ \boldsymbol{v}.\mathsf{rep}$

▷ Prove as claimed $\mathsf{INDEP}_3\mathsf{a}$: $\boldsymbol{v} \cong n$
$= \mathsf{phi\_iso}$

▷ Discharge $\mathsf{phi\_iso}, \mathsf{P}_2, \mathsf{P}_{2\text{-}2}, \mathsf{P}_{2\text{-}1}, \phi', \mathsf{phi'\_forms\_map}, \mathsf{P}_1, \mathsf{phi'\_fun}, \phi,$
$\mathsf{phi\_forms\_map}, \mathsf{phi\_fun}, \boldsymbol{v}, n$

▷ Prove INDEP_3 : $\{\Pi H : K.\mathsf{is\_fin\_dim\_over}\ L\}\ H.\mathsf{basis} \cong (\mathsf{dim}\ H)$
$= [\lambda H : K.\mathsf{is\_fin\_dim\_over}\ L]\ \mathsf{INDEP}_3\mathsf{a}\ (\mathsf{make}\ H.\mathsf{independence})$

▷ Discharge L, K, V, F

B.7.2.8 Results concerning spanning tuples

▷ Introduce F; V | F-space and J, K | subfield F

▷ Introduce W | subgroup V; n | $\mathbb{N}$ and $\boldsymbol{w}$: $W\hat{}n$

▷ Suppose H : $\{\forall x : K\}\ \{\forall v : W\}\ (x * v) \in W$

▷ We want to prove SPAN_3 : $(\boldsymbol{w}\text{-}\mathsf{span\_over}\ K) \subseteq W$

▷ Introduce v : $\boldsymbol{w}\text{-}\mathsf{span\_over}\ K$

▷ Let $\boldsymbol{x} = (v.\mathsf{ev}).\mathbf{_1}$: $K\hat{}n$

▷ Prove subresult P_1 : $(\boldsymbol{x} * \boldsymbol{w}) \in (W\hat{}n)$
$= [\lambda i]\ H\ (\boldsymbol{x}.\mathsf{ev}\ i)\ (\boldsymbol{w}.\mathsf{ev}\ i)$

▷ Prove subresult P_2 : $(\sum^*\ \boldsymbol{x}\ \boldsymbol{w}) \in W$
$= \mathsf{LINSUM}_1\ W\ \mathsf{P}_1$

▷ Discharge to prove as claimed SPAN_3 : $(\boldsymbol{w}\text{-}\mathsf{span\_over}\ K) \subseteq W$
using
$\mathsf{eq\_closed}\ (v.\mathsf{ev}).\mathbf{_2}\ \mathsf{P}_2 : v \in W$

▷ Discharge v

▷ Discharge $\mathsf{P}_2, \mathsf{P}_1, \boldsymbol{x}, H, \boldsymbol{w}, n, W$

▷ Introduce g_1, g_2 | subgroup_hom J and let $\mathsf{A} = g_1.\mathsf{agree}\ g_2$: subset F

▷ Suppose H_1 : $K \subseteq J$ and H_2 : $J.\mathsf{has\_fin\_span\_over}\ K$

▷ Suppose H_3 : $\{\forall v : J \cap \mathsf{A}\}\ \{\forall x : K\}\ (x \times v) \in \mathsf{A}$

▷ We want to prove $\mathsf{SPAN}_4 : (J \subseteq \mathsf{A}).\mathsf{or\_not}$

▷ Let $\mathsf{n} = \mathsf{size\_of\_span}\ H_2 : \mathbb{N}$

▷ Let $\boldsymbol{w} = \mathsf{spanning\_tuple}\ H_2 : J^{\wedge}\mathsf{n}$

▷ We want to prove subresult $\mathsf{P}_1 : (\boldsymbol{w} \in (\mathsf{A}^{\wedge}\mathsf{n})) \vee (\boldsymbol{w} \notin (\mathsf{A}^{\wedge}\mathsf{n}))$

▷ We want to prove subresult $\mathsf{P}_2 : (\boldsymbol{w} \in (\mathsf{A}^{\wedge}\mathsf{n})) \leftrightarrow (J \subseteq \mathsf{A})$

▷ Prove subresult $\mathsf{P}_{1\text{-}1} : \{\forall x : J\}\ (x \in \mathsf{A}) \vee (x \notin \mathsf{A})$
$= [\lambda x : J]\ \mathsf{FIELD}_1\ (g_1\ x)\ (g_2\ x)$

▷ Prove subresult $\mathsf{P}_{1\text{-}2} : \{\forall i : \mathsf{n}\}\ ((\boldsymbol{w}_{\backslash} i) \in \mathsf{A}) \vee ((\boldsymbol{w}_{\backslash} i) \notin \mathsf{A})$
$= [\lambda i : \mathsf{n}]\ \mathsf{P}_{1\text{-}1}\ (\mathsf{make}\ (\boldsymbol{w}.\mathsf{ev}\ i))$

▷ Prove subresult $\mathsf{P}_{1\text{-}3} : \mathsf{subset\_axioms}|\mathsf{n}\ ([\lambda i : \mathsf{n}]\ (\boldsymbol{w}_{\backslash} i) \in \mathsf{A})$
$= [\lambda i_1, i_2 : \mathsf{n}]\ [\lambda Q i : i_1 = i_2]\ [\lambda H : (\boldsymbol{w}_{\backslash} i_1) \in \mathsf{A}]\ \mathsf{eq\_closed}\ (\boldsymbol{w}.\mathsf{rep.resp}\ Q i)\ H$

▷ Let $\mathsf{P}_3 = ([\lambda i : \mathsf{n}]\ (\boldsymbol{w}_{\backslash} i) \in \mathsf{A}, \mathsf{P}_{1\text{-}3} : \mathsf{subset}\ \mathsf{n}) : \mathsf{subset}\ \mathsf{n}$

▷ Prove subresult as claimed $\mathsf{P}_1 : (\boldsymbol{w} \in (\mathsf{A}^{\wedge}\mathsf{n})) \vee (\boldsymbol{w} \notin (\mathsf{A}^{\wedge}\mathsf{n}))$
$= \mathsf{FIN}_6\ \mathsf{P}_3\ \mathsf{P}_{1\text{-}2}$

▷ We want to prove subresult $\mathsf{P}_{2\text{-}1} : (\boldsymbol{w} \in (\mathsf{A}^{\wedge}\mathsf{n})) \to J \subseteq \mathsf{A}$

▷ We want to prove subresult $\mathsf{P}_{2\text{-}2} : (J \subseteq \mathsf{A}) \to \boldsymbol{w} \in (\mathsf{A}^{\wedge}\mathsf{n})$

▷ Suppose $H_4 : J \subseteq \mathsf{A}$

▷ Discharge to prove subresult as claimed $\mathsf{P}_{2\text{-}2} : (J \subseteq \mathsf{A}) \to \boldsymbol{w} \in (\mathsf{A}^{\wedge}\mathsf{n})$
using
$[\lambda i : \mathsf{n}]\ H_4\ (\boldsymbol{w}.\mathsf{ev}\ i) : \boldsymbol{w} \in (\mathsf{A}^{\wedge}\mathsf{n})$

▷ Discharge H_4

▷ Suppose $H_5 : \boldsymbol{w} \in (\mathsf{A}^{\wedge}\mathsf{n})$

▷ Introduce $v : J$

▷ Prove subresult $\mathsf{P}_4 : v \in (\boldsymbol{w}\text{-}\mathsf{span\_over}\ K)$
$= H_{2\cdot\mathbf{2}\cdot\mathbf{2}}.\mathsf{snd.fst}\ v$

▷ Let $\boldsymbol{x} = \mathsf{P}_{4\cdot\mathbf{1}} : K^{\wedge}\mathsf{n}$

▷ Let $\boldsymbol{v} = \boldsymbol{x} * \boldsymbol{w} : F^{\wedge}\mathsf{n}$

▷ Prove subresult $\mathsf{P}_5 : (\sum \boldsymbol{v}) = v$
$= \mathsf{P}_{4\cdot\mathbf{2}}$

▷ Prove subresult $\mathsf{P}_6 : \boldsymbol{v} \in (J^{\wedge}\mathsf{n})$
$= [\lambda i : \mathsf{n}]\ \mathsf{times\_closed}\ (H_1\ (\boldsymbol{x}.\mathsf{ev}\ i))\ (\boldsymbol{w}.\mathsf{ev}\ i)$

▷ Prove subresult P_7 **:** $\{\forall g : \mathsf{subgroup\_hom}\ J\}\ (g\ (\sum \boldsymbol{v})) = (\sum\ (g \circ \boldsymbol{v}))$
$= [\lambda g : \mathsf{subgroup\_hom}\ J]\ \mathsf{LINSUM}_2\ J\ g\ \mathsf{P}_6$

▷ Prove subresult P_8 **:** $\boldsymbol{w} \in ((J \cap \mathsf{A})^\wedge \mathsf{n})$
$= [\lambda i : \mathsf{n}]\ \mathsf{pair}\ (\boldsymbol{w}.\mathsf{ev}\ i)\ (H_5\ i)$

▷ Prove subresult P_9 **:** $(g_1 \circ \boldsymbol{v}) = (g_2 \circ \boldsymbol{v})$
$= [\lambda i : \mathsf{n}]\ H_3\ (\mathsf{P}_8\ i)\ (\boldsymbol{x}.\mathsf{ev}\ i)$

▷ Prove subresult P_{10} **:** $(\sum \boldsymbol{v}) \in \mathsf{A}$
$= ((\mathsf{P}_7\ g_1).\mathsf{tran}\ (\sum.\mathsf{resp}\ \mathsf{P}_9)).\mathsf{tran}\ (\mathsf{P}_7\ g_2).\mathsf{symm}$

▷ Discharge to prove subresult as claimed $\mathsf{P}_{2\text{-}1}$ **:** $(\boldsymbol{w} \in (\mathsf{A}^\wedge \mathsf{n})) \to J \subseteq \mathsf{A}$
using
$\mathsf{eq\_closed}\ \mathsf{P}_5\ \mathsf{P}_{10}$ **:** $v \in \mathsf{A}$

▷ Discharge v, H_5

▷ Prove subresult as claimed P_2 **:** $(\boldsymbol{w} \in (\mathsf{A}^\wedge \mathsf{n})) \leftrightarrow (J \subseteq \mathsf{A})$
$= \mathsf{pair}\ \mathsf{P}_{2\text{-}1}\ \mathsf{P}_{2\text{-}2}$

▷ Prove as claimed SPAN_4 **:** $(J \subseteq \mathsf{A}).\mathsf{or\_not}$
$= \mathsf{DEC}_2\ \mathsf{P}_2\ \mathsf{P}_1$

▷ Discharge $\mathsf{P}_2, \mathsf{P}_{2\text{-}1}, \mathsf{P}_{10}, \mathsf{P}_9, \mathsf{P}_8, \mathsf{P}_7, \mathsf{P}_6, \mathsf{P}_5, \boldsymbol{v}, \boldsymbol{x}, \mathsf{P}_4, \mathsf{P}_{2\text{-}2}, \mathsf{P}_1, \mathsf{P}_3, \mathsf{P}_{1\text{-}3}, \mathsf{P}_{1\text{-}2}, \mathsf{P}_{1\text{-}1}$,
$\boldsymbol{w}, \mathsf{n}, H_3, H_2, H_1, \mathsf{A}, g_2, g_1$

▷ Introduce $m, n \mid \mathbb{N}$; $\boldsymbol{v}$ **:** m-$\mathsf{tuple\_independent\_over}\ K\ V$ and $\boldsymbol{w}$
: n-$\mathsf{tuple\_independent\_over}\ K\ V$

These results were assumed without proof due to a lack of time. The proofs are straightforward but very tedious inductions.

▷ Assume without proof SPAN_5
: $((\boldsymbol{v}\text{-}\mathsf{span\_over}\ K) \subseteq (\boldsymbol{w}\text{-}\mathsf{span\_over}\ K)) \to \boldsymbol{v} \preccurlyeq \boldsymbol{w}$

▷ Assume without proof SPAN_6 **:** $((\boldsymbol{v}\text{-}\mathsf{span\_over}\ K) \subseteq (\boldsymbol{w}\text{-}\mathsf{span\_over}\ K)) \to$
$(\boldsymbol{v} \cong \boldsymbol{w}) \to (\boldsymbol{w}\text{-}\mathsf{span\_over}\ K) \subseteq (\boldsymbol{v}\text{-}\mathsf{span\_over}\ K)$

▷ Discharge $\boldsymbol{w}, \boldsymbol{v}, n, m, K, J, V, F$

B.7.2.9 Results about finite dimensional vectorspaces

▷ Introduce F and $V \mid F$-space

▷ Introduce L

▷ We want to prove $\mathsf{FINDIM_3b} : L.\mathsf{has\_basis\_over}\ L\ (\mathsf{one\_tuple}\ \mathbf{1})$

▷ Prove subresult $\mathsf{FINDIM_3b_1} : (\mathsf{one\_tuple}\ \mathbf{1}) \in (L\hat{}1)$
$= [\lambda\_ : 1]\ \mathsf{un\_closed}|F|L$

▷ We want to prove subresult $\mathsf{FINDIM_3b_{2\_1}} : L \subseteq ((\mathsf{one\_tuple}\ \mathbf{1})\text{-}\mathsf{span\_over}\ L)$

▷ Introduce $v : L$

▷ Let $\boldsymbol{x} = (\mathsf{one\_tuple}\ v.\mathsf{rep}, [\lambda\_ : 1]\ v.\mathsf{ev} : L\hat{}1) : L\hat{}1$

▷ Prove subresult $\mathsf{Q_{2\_1}} : ((v.\mathsf{rep} \times \mathbf{1}) + \mathbf{0}) = (v.\mathsf{rep} \times \mathbf{1})$
$= (v.\mathsf{rep} \times \mathbf{1}).\mathsf{plus\_ze}$

▷ Prove subresult $\mathsf{Q_{2\_2}} : (v.\mathsf{rep} \times \mathbf{1}) = v.\mathsf{rep}$
$= v.\mathsf{rep.times\_un}$

▷ Discharge to prove subresult as claimed $\mathsf{FINDIM_3b_{2\_1}}$
$: L \subseteq ((\mathsf{one\_tuple}\ \mathbf{1})\text{-}\mathsf{span\_over}\ L)$
using
$(\boldsymbol{x}, \mathsf{Q_{2\_1}.tran}\ \mathsf{Q_{2\_2}} : v \in ((\mathsf{one\_tuple}\ \mathbf{1})\text{-}\mathsf{span\_over}\ L))$
$: v \in ((\mathsf{one\_tuple}\ \mathbf{1})\text{-}\mathsf{span\_over}\ L)$

▷ Discharge v

▷ We want to prove subresult $\mathsf{FINDIM_3b_{2\_2}} : ((\mathsf{one\_tuple}\ \mathbf{1})\text{-}\mathsf{span\_over}\ L) \subseteq L$

▷ Introduce $v : (\mathsf{one\_tuple}\ \mathbf{1})\text{-}\mathsf{span\_over}\ L$

▷ Let $\boldsymbol{y} = (v.\mathsf{ev})._{\mathbf{1}} : L\hat{}1$

▷ Let $\mathsf{x} = \boldsymbol{y}.\mathsf{ev}\ (\underline{0}\ 0) : L$

▷ Prove subresult $\mathsf{P_1} : (\mathsf{x} \times \mathbf{1}) \in L$
$= \mathsf{times\_closed\ x.ev\ un\_closed}$

▷ Prove subresult $\mathsf{P_2} : (\sum^* \boldsymbol{y}\ (\mathsf{one\_tuple}\ \mathbf{1})) \in L$
$= \mathsf{plus\_closed}\ \mathsf{P_1}\ \mathsf{ze\_closed}$

▷ Discharge to prove subresult as claimed $\mathsf{FINDIM_3b_{2\_2}}$
$: ((\mathsf{one\_tuple}\ \mathbf{1})\text{-}\mathsf{span\_over}\ L) \subseteq L$
using
$\mathsf{eq\_closed}\ (v.\mathsf{ev})._{\mathbf{2}}\ \mathsf{P_2} : v \in L$

▷ Discharge v

▷ Prove subresult $\mathsf{FINDIM_3b_2} : (\mathsf{one\_tuple}\ \mathbf{1}).\mathsf{spans}\ L\ L$
$= \mathsf{pair}\ \mathsf{FINDIM_3b_{2\_1}}\ \mathsf{FINDIM_3b_{2\_2}}$

▷ We want to prove subresult $\mathsf{FINDIM_3b_3}$
$: (\mathsf{one\_tuple}\ \mathbf{1}) \in (1\text{-}\mathsf{tuple\_independent\_over}\ L\ F)$

▷ Introduce $\boldsymbol{z}$ **:** $L^{\wedge}1$

▷ Suppose Q **:** $(\sum^{*}\ \boldsymbol{z}\ (\mathsf{one\_tuple}\ \mathbf{1})) = \mathbf{0}$

▷ Introduce i **:** 1

▷ Prove subresult $\mathsf{Q}_{3\text{–}0}$ **:** $\mathsf{equal\_in}\ 1\ i\ (\underline{0}\ 0)$
$= (\mathsf{CANON}_3\ i).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_{3\text{–}1}$ **:** $(\boldsymbol{z}_{\backslash} i) = (\boldsymbol{z}_{\backslash}(\underline{0}\ 0))$
$= \boldsymbol{z}.\mathsf{rep.resp}\ \mathsf{Q}_{3\text{–}0}$

▷ Prove subresult $\mathsf{Q}_{3\text{–}2}$ **:** $(\boldsymbol{z}_{\backslash}(\underline{0}\ 0)) = ((\boldsymbol{z}_{\backslash}(\underline{0}\ 0)) \times \mathbf{1})$
$= (\boldsymbol{z}_{\backslash}(\underline{0}\ 0)).\mathsf{times\_un.symm}$

▷ Prove subresult $\mathsf{Q}_{3\text{–}3}$ **:** $((\boldsymbol{z}_{\backslash}(\underline{0}\ 0)) \times \mathbf{1}) = (((\boldsymbol{z}_{\backslash}(\underline{0}\ 0)) \times \mathbf{1}) + \mathbf{0})$
$= ((\boldsymbol{z}_{\backslash}(\underline{0}\ 0)) \times \mathbf{1}).\mathsf{plus\_ze.symm}$

▷ Discharge to prove subresult as claimed $\mathsf{FINDIM}_3\mathsf{b}_3$
: $(\mathsf{one\_tuple}\ \mathbf{1}) \in (1\text{-}\mathsf{tuple\_independent\_over}\ L\ F)$
using
$(\mathsf{Q}_{3\text{–}1}.\mathsf{tran}\ \mathsf{Q}_{3\text{–}2}).\mathsf{tran}\ (\mathsf{Q}_{3\text{–}3}.\mathsf{tran}\ Q)$ **:** $(\boldsymbol{z}_{\backslash} i) = \mathbf{0}$

▷ Discharge $i, Q, \boldsymbol{z}$

▷ Prove as claimed $\mathsf{FINDIM}_3\mathsf{b}$ **:** $L.\mathsf{has\_basis\_over}\ L\ (\mathsf{one\_tuple}\ \mathbf{1})$
$= \mathsf{pair}_3\ \mathsf{FINDIM}_3\mathsf{b}_1\ \mathsf{FINDIM}_3\mathsf{b}_2\ \mathsf{FINDIM}_3\mathsf{b}_3$

▷ Prove $\mathsf{FINDIM}_3\mathsf{a}$ **:** $L.\mathsf{has\_fin\_dim\_over}\ L\ 1$
$= (\mathsf{one\_tuple}\ \mathbf{1}, \mathsf{FINDIM}_3\mathsf{b} : L.\mathsf{has\_fin\_dim\_over}\ L\ 1)$

▷ Prove FINDIM_3 **:** $L.\mathsf{is\_fin\_dim\_over}\ L$
$= (1, \mathsf{FINDIM}_3\mathsf{a} : L.\mathsf{is\_fin\_dim\_over}\ L)$

▷ Discharge $\mathsf{FINDIM}_3\mathsf{b}_3, \mathsf{Q}_{3\text{–}3}, \mathsf{Q}_{3\text{–}2}, \mathsf{Q}_{3\text{–}1}, \mathsf{Q}_{3\text{–}0}, \mathsf{FINDIM}_3\mathsf{b}_2, \mathsf{FINDIM}_3\mathsf{b}_{2\text{–}2}, \mathsf{P}_2, \mathsf{P}_1,$
$\mathsf{x}, \boldsymbol{y}, \mathsf{FINDIM}_3\mathsf{b}_{2\text{–}1}, \mathsf{Q}_{2\text{–}2}, \mathsf{Q}_{2\text{–}1}, \boldsymbol{x}, \mathsf{FINDIM}_3\mathsf{b}_1, L$

▷ Introduce K | $\mathsf{subgroup}\ V$ and L | $\mathsf{subfield}\ F$

▷ Introduce m, n | $\mathbb{N}$; $\boldsymbol{v}$ **:** $V^{\wedge}m$ and $\boldsymbol{w}$ **:** $V^{\wedge}n$

▷ Suppose H_1 **:** $K.\mathsf{has\_basis\_over}\ L\ \boldsymbol{v}$

▷ Suppose H_2 **:** $K.\mathsf{has\_basis\_over}\ L\ \boldsymbol{w}$

▷ Prove subresult Q **:** $(\boldsymbol{v}\text{-}\mathsf{span\_over}\ L) = (\boldsymbol{w}\text{-}\mathsf{span\_over}\ L)$
$= H_1.\mathsf{snd}_3.\mathsf{equal\_subs\_symm.equal\_subs\_tran}\ H_2.\mathsf{snd}_3$

▷ Prove subresult P_1 **:** $\boldsymbol{v} \preccurlyeq \boldsymbol{w}$
$= \mathsf{SPAN}_5\ H_1.\mathsf{thd}_3\ H_2.\mathsf{thd}_3\ \mathsf{Q.fst}$

▷ Prove subresult $\mathsf{P}_2 : \boldsymbol{w} \preccurlyeq \boldsymbol{v}$
= $\mathsf{SPAN}_5\ H_2.\mathsf{thd}_3\ H_1.\mathsf{thd}_3\ \mathsf{Q.snd}$

▷ Prove $\mathsf{FINDIM}_4\mathsf{b} : \boldsymbol{v} \cong \boldsymbol{w}$
= $\mathsf{smaller\_asymm}\ \mathsf{P}_2\ \mathsf{P}_1$

▷ Discharge $\mathsf{P}_2, \mathsf{P}_1, \mathsf{Q}, H_2, H_1, \boldsymbol{w}, \boldsymbol{v}$

▷ Suppose $H_1 : K.\mathsf{has\_fin\_dim\_over}\ L\ m$

▷ Suppose $H_2 : K.\mathsf{has\_fin\_dim\_over}\ L\ n$

▷ Let $\boldsymbol{v} = H_{1\cdot\mathbf{1}} : V^{\wedge}m$ and $\boldsymbol{w} = H_{2\cdot\mathbf{1}} : V^{\wedge}n$

▷ Prove subresult $\mathsf{Q} : \boldsymbol{v} \cong \boldsymbol{w}$
= $\mathsf{FINDIM}_4\mathsf{b}\ \boldsymbol{v}\ \boldsymbol{w}\ H_{1\cdot\mathbf{2}}\ H_{2\cdot\mathbf{2}}$

▷ Prove subresult $\mathsf{Q}_1 : \boldsymbol{v} \cong m$
= $\mathsf{INDEP}_3\mathsf{a}\ H_{1\cdot\mathbf{2}}.\mathsf{thd}_3$

▷ Prove subresult $\mathsf{Q}_2 : \boldsymbol{w} \cong n$
= $\mathsf{INDEP}_3\mathsf{a}\ H_{2\cdot\mathbf{2}}.\mathsf{thd}_3$

▷ Prove subresult $\mathsf{Q}_3 : m \cong n$
= $\mathsf{Q}_1.\mathsf{eqsize\_symm.eqsize\_tran}\ (\mathsf{Q.eqsize\_tran}\ \mathsf{Q}_2)$

▷ Prove $\mathsf{FINDIM}_4\mathsf{a} : m = n$
= $\mathsf{EQSIZE}_1\ \mathsf{Q}_3$

▷ Discharge $\mathsf{Q}_3, \mathsf{Q}_2, \mathsf{Q}_1, \mathsf{Q}, \boldsymbol{w}, \boldsymbol{v}, H_2, H_1, n, m$

▷ Prove $\mathsf{FINDIM}_4 : \{\Pi H_1, H_2 : K.\mathsf{is\_fin\_dim\_over}\ L\}\ (\mathsf{dim}\ H_1) = (\mathsf{dim}\ H_2)$
= $[\lambda H_1, H_2 : K.\mathsf{is\_fin\_dim\_over}\ L]\ \mathsf{FINDIM}_4\mathsf{a}\ H_{1\cdot\mathbf{2}}\ H_{2\cdot\mathbf{2}}$

▷ Discharge L, K, V, F

B.8 Galois theory: definitions and setting

▷ Introduce F | field

B.8.1 Subfield morphisms

B.8.1.1 Definition of the automorphisms of a subfield

▷ Introduce L : subfield F

▷ Define resp_plus_times $=$ (resp_map$_2$ $+$ L) $\cap$ (resp_map$_2$ $\times$ L)

: subset (map F F)

▷ Define Aut_subset $=$ (Perm L) $\cap$ resp_plus_times.as_a_subset_of_perm

: subset (perm F)

▷ Let coercion Aut_rep $= [\lambda g : \text{Aut\_subset}]\ g.\text{rep}$: Aut_subset $\to$ perm F

▷ Introduce g : Aut_subset

▷ Introduce g_1, g_2 : Aut_subset

▷ Introduce x, y : L

▷ Prove subresult P$_1$: $(g_2\ x) \in L$

$=$ PERMc g_2.ev.fst x

▷ Prove subresult P$_2$: $(g_2\ y) \in L$

$=$ PERMc g_2.ev.fst y

▷ Prove subresult P$_3$: $(g^{-1}\ x) \in L$

$=$ PERMc g.ev.fst.inv_closed x

▷ Prove subresult P$_4$: $(g^{-1}\ y) \in L$

$=$ PERMc g.ev.fst.inv_closed y

▷ Prove subresult id_P$_1$: id $\in$ (Perm L)

$=$ id_closed|(perm $F.\mathbf{1}.\mathbf{1}$)|(Perm L)

▷ Prove subresult id_P$_2$: id $\in$ resp_plus_times.as_a_subset_of_perm

$=$ pair ($[\lambda x, y : L]$ refl $(x + y)$) ($[\lambda x, y : L]$ refl $(x \times y)$)

▷ Prove subresult id_in_Aut : id $\in$ Aut_subset

$=$ pair id_P$_1$ id_P$_2$

▷ Prove subresult o_P$_1$: $(g_1 \circ g_2) \in$ (Perm L)

$=$ o_closed g_1.ev.fst g_2.ev.fst

▷ Prove subresult o_Q$_1$: $(g_1 \circ g_2\ (x + y)) = (g_1\ (g_2\ (x + y)))$

$=$ rewrite_o g_1 g_2 $(x + y)$

▷ Prove subresult o_Q$_2$: $(g_1\ (g_2.\text{rep}\ (x + y))) = (g_1\ ((g_2.\text{rep}\ x) + (g_2.\text{rep}\ y)))$

$=$ g_1.resp (g_2.ev.snd.fst x y)

▷ Prove subresult $\mathsf{o\_Q_3} : (g_1\ ((g_2\ x) + (g_2\ y))) = ((g_1\ (g_2\ x)) + (g_1\ (g_2\ y)))$
$= g_1.\mathsf{ev.snd.fst}\ \mathsf{P}_1\ \mathsf{P}_2$

▷ Prove subresult $\mathsf{o\_Q_{4\text{-}1}} : (g_1\ (g_2\ x))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (g_1 \circ g_2\ x)$
$= (\mathsf{rewrite\_o}\ g_1\ g_2\ x).\mathsf{symm}$

▷ Prove subresult $\mathsf{o\_Q_{4\text{-}2}} : (g_1\ (g_2\ y))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (g_1 \circ g_2\ y)$
$= (\mathsf{rewrite\_o}\ g_1\ g_2\ y).\mathsf{symm}$

▷ Prove subresult $\mathsf{o\_Q_4}$
$: ((g_1\ (g_2\ x)) + (g_1\ (g_2\ y))) = ((g_1 \circ g_2\ x) + (g_1 \circ g_2\ y))$
$= \mathsf{plus\_resp}\ \mathsf{o\_Q_{4\text{-}1}}\ \mathsf{o\_Q_{4\text{-}2}}$

▷ Prove subresult $\mathsf{o\_P_{2\text{-}1}}$
$: (g_1 \circ g_2\ (x + y))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ ((g_1 \circ g_2\ x) + (g_1 \circ g_2\ y))$
$= (\mathsf{o\_Q_1.tran\ o\_Q_2}).\mathsf{tran}\ (\mathsf{o\_Q_3.tran\ o\_Q_4})$

▷ Prove subresult $\mathsf{o\_Q_5} : (g_1 \circ g_2\ (x \times y)) = (g_1\ (g_2\ (x \times y)))$
$= \mathsf{rewrite\_o}\ g_1\ g_2\ (x \times y)$

▷ Prove subresult $\mathsf{o\_Q_6} : (g_1\ (g_2.\mathsf{rep}\ (x \times y))) = (g_1\ ((g_2.\mathsf{rep}\ x) \times (g_2.\mathsf{rep}\ y)))$
$= g_1.\mathsf{resp}\ (g_2.\mathsf{ev.snd.snd}\ x\ y)$

▷ Prove subresult $\mathsf{o\_Q_7} : (g_1\ ((g_2\ x) \times (g_2\ y))) = ((g_1\ (g_2\ x)) \times (g_1\ (g_2\ y)))$
$= g_1.\mathsf{ev.snd.snd}\ \mathsf{P}_1\ \mathsf{P}_2$

▷ Prove subresult $\mathsf{o\_Q_8}$
$: ((g_1\ (g_2\ x)) \times (g_1\ (g_2\ y))) = ((g_1 \circ g_2\ x) \times (g_1 \circ g_2\ y))$
$= \mathsf{times\_resp}\ \mathsf{o\_Q_{4\text{-}1}}\ \mathsf{o\_Q_{4\text{-}2}}$

▷ Prove subresult $\mathsf{o\_P_{2\text{-}2}}$
$: (g_1 \circ g_2\ (x \times y))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ ((g_1 \circ g_2\ x) \times (g_1 \circ g_2\ y))$
$= (\mathsf{o\_Q_5.tran\ o\_Q_6}).\mathsf{tran}\ (\mathsf{o\_Q_7.tran\ o\_Q_8})$

▷ Prove subresult $\mathsf{inv\_P_1} : g^{-1} \in (\mathsf{Perm}\ L)$
$= \mathsf{inv\_closed}\ g.\mathsf{ev.fst}$

▷ Prove subresult $\mathsf{inv\_Q_{1\text{-}1\text{-}1\text{-}1}} : ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ x)\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (((g \circ g^{-1}))_{\cdot\mathbf{1}\cdot\mathbf{1}}\ x)$
$= (g.\mathsf{o\_inv}\ x).\mathsf{symm}$

▷ Prove subresult $\mathsf{inv\_Q_{1\text{-}1\text{-}1\text{-}2}} : (g \circ g^{-1}\ x) = (g\ (g^{-1}\ x))$
$= \mathsf{rewrite\_o}\ g\ g^{-1}\ x$

▷ Prove subresult $\mathsf{inv\_Q_{1\text{-}1\text{-}1}} : ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ x)\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (g\ (g^{-1}\ x))$
$= \mathsf{inv\_Q_{1\text{-}1\text{-}1\text{-}1.tran\ inv\_Q_{1\text{-}1\text{-}1\text{-}2}}}$

▷ Prove subresult $\mathsf{inv\_Q_{1\text{-}1\text{-}2\text{-}1}} : ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ y)\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (((g \circ g^{-1}))_{\cdot\mathbf{1}\cdot\mathbf{1}}\ y)$
$= (g.\mathsf{o\_inv}\ y).\mathsf{symm}$

▷ Prove subresult $\mathsf{inv\_Q_{1\text{-}1\text{-}2\text{-}2}} : (g \circ g^{-1}\ y) = (g\ (g^{-1}\ y))$
$= \mathsf{rewrite\_o}\ g\ g^{-1}\ y$

▷ Prove subresult $\mathsf{inv\_Q_{1\text{-}1\text{-}2}} : ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ y)\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (g\ (g^{-1}\ y))$
$= \mathsf{inv\_Q_{1\text{-}1\text{-}2\text{-}1}.tran\ inv\_Q_{1\text{-}1\text{-}2\text{-}2}}$

▷ Prove subresult $\mathsf{inv\_Q_{1\text{-}1}}$
$: (((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ x) + ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ y)) = ((g\ (g^{-1}\ x)) + (g\ (g^{-1}\ y)))$
$= \mathsf{plus\_resp\ inv\_Q_{1\text{-}1\text{-}1}\ inv\_Q_{1\text{-}1\text{-}2}}$

▷ Prove subresult $\mathsf{inv\_Q_{1\text{-}2}}$
$: ((g\ (g^{-1}\ x)) + (g\ (g^{-1}\ y))) = (g\ ((g^{-1}\ x) + (g^{-1}\ y)))$
$= (g.\mathsf{ev.snd.fst}\ \mathsf{P}_3\ \mathsf{P}_4).\mathsf{symm}$

▷ Prove subresult $\mathsf{inv\_Q_1}$
$: (g^{-1}\ (((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ x) + ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ y))) = (g^{-1}\ (g\ ((g^{-1}\ x) + (g^{-1}\ y))))$
$= g^{-1}.\mathsf{resp}\ (\mathsf{inv\_Q_{1\text{-}1}.tran\ inv\_Q_{1\text{-}2}})$

▷ Prove subresult $\mathsf{inv\_Q_2}$
$: (g^{-1}\ (g\ ((g^{-1}\ x) + (g^{-1}\ y))))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (g^{-1} \circ g\ ((g^{-1}\ x) + (g^{-1}\ y)))$
$= (\mathsf{rewrite\_o}\ g^{-1}\ g\ ((g^{-1}\ x) + (g^{-1}\ y))).\mathsf{symm}$

▷ Prove subresult $\mathsf{inv\_Q_3} : (g^{-1} \circ g\ ((g^{-1}\ x) + (g^{-1}\ y))) = ((g^{-1}\ x) + (g^{-1}\ y))$
$= \mathsf{inv\_o}\ g\ ((g^{-1}\ x) + (g^{-1}\ y))$

▷ Prove subresult $\mathsf{inv\_P_{2\text{-}1}}$
$: (g^{-1}\ (((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ x) + ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ y)))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ ((g^{-1}\ x) + (g^{-1}\ y))$
$= (\mathsf{inv\_Q_1.tran\ inv\_Q_2}).\mathsf{tran\ inv\_Q_3}$

▷ Prove subresult $\mathsf{inv\_Q_{4\text{-}1}}$
$: (((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ x) \times ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ y)) = ((g\ (g^{-1}\ x)) \times (g\ (g^{-1}\ y)))$
$= \mathsf{times\_resp\ inv\_Q_{1\text{-}1\text{-}1}\ inv\_Q_{1\text{-}1\text{-}2}}$

▷ Prove subresult $\mathsf{inv\_Q_{4\text{-}2}}$
$: ((g\ (g^{-1}\ x)) \times (g\ (g^{-1}\ y))) = (g\ ((g^{-1}\ x) \times (g^{-1}\ y)))$
$= (g.\mathsf{ev.snd.snd}\ \mathsf{P}_3\ \mathsf{P}_4).\mathsf{symm}$

▷ Prove subresult $\mathsf{inv\_Q_4}$
$: (g^{-1}\ (((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ x) \times ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ y))) = (g^{-1}\ (g\ ((g^{-1}\ x) \times (g^{-1}\ y))))$
$= g^{-1}.\mathsf{resp}\ (\mathsf{inv\_Q_{4\text{-}1}.tran\ inv\_Q_{4\text{-}2}})$

▷ Prove subresult $\mathsf{inv\_Q_5}$
$: (g^{-1}\ (g\ ((g^{-1}\ x) \times (g^{-1}\ y))))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (g^{-1} \circ g\ ((g^{-1}\ x) \times (g^{-1}\ y)))$
$= (\mathsf{rewrite\_o}\ g^{-1}\ g\ ((g^{-1}\ x) \times (g^{-1}\ y))).\mathsf{symm}$

▷ Prove subresult $\mathsf{inv\_Q_6} : (g^{-1} \circ g\ ((g^{-1}\ x) \times (g^{-1}\ y))) = ((g^{-1}\ x) \times (g^{-1}\ y))$
$= \mathsf{inv\_o}\ g\ ((g^{-1}\ x) \times (g^{-1}\ y))$

▷ Prove subresult $\mathsf{inv\_P_{2\text{-}2}}$
: $(g^{-1}\ (((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ x) \times ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ y)))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ ((g^{-1}\ x) \times (g^{-1}\ y))$
$= (\mathsf{inv\_Q_4.tran\ inv\_Q_5}).\mathsf{tran\ inv\_Q_6}$

▷ Discharge y, x

▷ Prove subresult $\mathsf{o\_P_2}$: $(g_1 \circ g_2) \in$ resp_plus_times.as_a_subset_of_perm
$= \mathsf{pair\ o\_P_{2\text{-}1}\ o\_P_{2\text{-}2}}$

▷ Prove subresult $\mathsf{o\_in\_Aut}$: $(g_1 \circ g_2) \in$ Aut_subset
$= \mathsf{pair\ o\_P_1\ o\_P_2}$

▷ Prove subresult $\mathsf{inv\_P_2}$: $g^{-1} \in$ resp_plus_times.as_a_subset_of_perm
$= \mathsf{pair\ inv\_P_{2\text{-}1}\ inv\_P_{2\text{-}2}}$

▷ Prove subresult $\mathsf{inv\_in\_Aut}$: $g^{-1} \in$ Aut_subset
$= \mathsf{pair\ inv\_P_1\ inv\_P_2}$

▷ Discharge g_2, g_1, g

▷ Prove Aut_forms_subgroup : subgroup_axioms Aut_subset
$= \mathsf{pair_3\ id\_in\_Aut\ o\_in\_Aut\ inv\_in\_Aut}$

▷ Define Aut $=$ (Aut_subset, Aut_forms_subgroup : subgroup (perm F))
: subgroup (perm F)

▷ Discharge $\mathsf{inv\_in\_Aut, inv\_P_2, o\_in\_Aut, o\_P_2, inv\_P_{2\text{-}2}, inv\_Q_6, inv\_Q_5, inv\_Q_4,}$
$\mathsf{inv\_Q_{4\text{-}2}, inv\_Q_{4\text{-}1}, inv\_P_{2\text{-}1}, inv\_Q_3, inv\_Q_2, inv\_Q_1, inv\_Q_{1\text{-}2}, inv\_Q_{1\text{-}1},}$
$\mathsf{inv\_Q_{1\text{-}1\text{-}2}, inv\_Q_{1\text{-}1\text{-}2\text{-}2}, inv\_Q_{1\text{-}1\text{-}2\text{-}1}, inv\_Q_{1\text{-}1\text{-}1}, inv\_Q_{1\text{-}1\text{-}1\text{-}2}, inv\_Q_{1\text{-}1\text{-}1\text{-}1},}$
$\mathsf{inv\_P_1, o\_P_{2\text{-}2}, o\_Q_8, o\_Q_7, o\_Q_6, o\_Q_5, o\_P_{2\text{-}1}, o\_Q_4, o\_Q_{4\text{-}2}, o\_Q_{4\text{-}1}, o\_Q_3,}$
$\mathsf{o\_Q_2, o\_Q_1, o\_P_1, id\_in\_Aut, id\_P_2, id\_P_1, P_4, P_3, P_2, P_1, Aut\_rep}$

▷ Define coercion $\mathsf{Aut\_Perm} = [\lambda g : \mathsf{Aut}]\ g.\mathsf{ev.fst}$: Aut $\to$ Perm L

▷ Introduce g : Aut

▷ Prove subresult $\mathsf{P_1}$: $(g^{-1}\ \mathbf{1}) \in L$
$= \mathsf{PERMc}\ g.\mathsf{inv\_closed}\ (\mathsf{un\_closed}|F|L)$

▷ Prove subresult $\mathsf{Q_1}$: $\mathbf{1} = (g \circ g^{-1}\ \mathbf{1})$
$= (g.\mathsf{o\_inv}\ \mathbf{1}).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q_2}$: $(g \circ g^{-1}\ \mathbf{1}) = (g\ (g^{-1}\ \mathbf{1}))$
$= \mathsf{rewrite\_o}\ g\ g^{-1}\ \mathbf{1}$

▷ Prove subresult $\mathsf{P_2}$: $\neg(\{\forall x : L\}\ (g\ x) = \mathbf{0})$
$= [\lambda H : \{\forall x : L\}\ (g\ x) = \mathbf{0}]\ \mathsf{nontriv}\ ((\mathsf{Q_1.tran\ Q_2}).\mathsf{tran}\ (H\ \mathsf{P_1}))$

▷ Prove Aut_is_subfield_hom : g.rep.rep ∈ (subfield_hom L)
= pair g.ev.snd.fst (pair P_2 g.ev.snd.snd)

▷ Discharge $\mathsf{P}_2, \mathsf{Q}_2, \mathsf{Q}_1, \mathsf{P}_1, g$

▷ Define coercion Aut_hom = $[\lambda g : \mathsf{Aut}]$
(g, Aut_is_subfield_hom g : subfield_hom L) : Aut → subfield_hom L

▷ Discharge L

B.8.1.2 Subfield morphisms that fix a second subfield

▷ Allow -fix_hom to be written postfix

▷ Define -fix_hom
= $[\lambda K, J : \mathsf{subfield}\ F]$ (map_from J) ∩ ((resp_plus_times J) ∩ (fixing K))
: (subfield F) → (subfield F) → subset (map F F)

B.8.1.3 Definition of fixing subgroups and fixed subfields

▷ Introduce L : subfield F

▷ Introduce K : subfield F

▷ Define fixing_subset = (fixing K).as_a_subset_of_perm : subset (perm $F_{\cdot\mathbf{1}\cdot\mathbf{1}}$)

▷ Let coercion fixing_rep = $[\lambda g : \mathsf{fixing\_subset}]$ g.rep : fixing_subset → perm F

▷ Introduce g_1, g_2 : fixing_subset and x : K

▷ Prove subresult id_is_fixing : id.fixes x
= refl x

▷ Prove subresult Q_1 : $(g_1 \circ g_2\ x) = (g_1\ (g_2\ x))$
= rewrite_o g_1 g_2 x

▷ Prove subresult P_1 : $(g_2\ x) \in K$
= eq_closed (g_2.ev x).symm x.ev

▷ Prove subresult Q_2 : g_1.fixes (g_2 x)
= g_1.ev P_1

▷ Prove subresult Q_3 : g_2.fixes x
= g_2.ev x

▷ Prove subresult o_is_fixing : $(g_1 \circ g_2)$.fixes x
= Q_1.tran (Q_2.tran Q_3)

▷ Prove subresult Q_4 : $({g_1}^{-1}\ x) = ({g_1}^{-1}\ (g_1\ x))$
= ${g_1}^{-1}$.resp (g_1.ev x).symm

▷ Prove subresult Q_5 : $({g_1}^{-1}\ (g_1\ x))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ ({g_1}^{-1} \circ g_1\ x)$
= (rewrite_o ${g_1}^{-1}\ g_1\ x$).symm

▷ Prove subresult Q_6 : $({g_1}^{-1} \circ g_1)$.fixes x
= inv_o $g_1\ x$

▷ Prove subresult inv_is_fixing : ${g_1}^{-1}$.fixes x
= Q_4.tran (Q_5.tran Q_6)

▷ Discharge x, g_2, g_1

▷ Prove fixing_forms_subgroup : subgroup_axioms fixing_subset
= pair_3 id_is_fixing o_is_fixing inv_is_fixing

▷ Define fixing_group
= (fixing_subset, fixing_forms_subgroup : subgroup (perm F))
: subgroup (perm F)

▷ Prove aut_forms_subgroup : (fixing_group ∩ (Aut L)) ⊆ (Aut L)
= [λg : fixing_group ∩ (Aut L)] g.ev.snd

▷ Define aut
= (fixing_group ∩ (Aut L), aut_forms_subgroup : subgroup_of (Aut L))
: subgroup_of (Aut L)

▷ Discharge
inv_is_fixing, Q_6, Q_5, Q_4, o_is_fixing, Q_3, Q_2, P_1, Q_1, id_is_fixing, fixing_rep, K

▷ Introduce G : subgroup (perm F)

▷ Define is_fixed = $[\lambda x : F]\ (\{\forall g : G\}\ g$.fixes $x) \wedge (x \in L)$: $F \to$ prop

▷ Prove fixed_forms_subset : subset_axioms is_fixed
= $[\lambda x_1, x_2 \mid F]\ [\lambda Q : x_1 = x_2]\ [\lambda H : x_1$.is_fixed]
pair ($[\lambda g : G]\ (g$.resp Q.symm).tran ((H.fst g).tran Q)) (eq_closed $Q\ H$.snd)

▷ Define fixed_subset = (is_fixed, fixed_forms_subset : subset F) : subset F

▷ Discharge G

▷ Introduce G : subgroup_of (Aut L)

▷ Introduce x, y : fixed_subset G; g : G and let gg = $G_{\cdot\mathbf{2}}\ g$: Aut L

▷ Prove subresult ze_is_fixed : g.fixes $\mathbf{0}$
= gg.on_ze

▷ Prove subresult Q_1 : $(g\ (x + y)) = ((g\ x) + (g\ y))$
= gg.on_plus x.ev.snd y.ev.snd

▷ Prove subresult Q_2 : $((g\ x) + (g\ y)) = (x + y)$
= plus_resp $(x$.ev.fst $g)$ $(y$.ev.fst $g)$

▷ Prove subresult plus_is_fixed : g.fixes $(x + y)$
= Q_1.tran Q_2

▷ Prove subresult Q_3 : $(g\ (-x)) = (-(g\ x))$
= gg.on_neg x.ev.snd

▷ Prove subresult Q_4 : $(-(g\ x)) = (-x)$
= neg_resp $(x$.ev.fst $g)$

▷ Prove subresult neg_is_fixed : g.fixes $(-x)$
= Q_3.tran Q_4

▷ Discharge g, y, x

▷ Prove subresult P_ze : $\mathbf{0} \in$ (fixed_subset G)
= pair ze_is_fixed (ze_closed$|F|L$)

▷ Prove subresult P_plus : $\{\forall x, y :$ fixed_subset $G\}$ $(x + y) \in$ (fixed_subset G)
= $[\lambda x, y :$ fixed_subset $G]$ pair (plus_is_fixed x y) (plus_closed x.ev.snd y.ev.snd)

▷ Prove subresult P_neg : $\{\forall x :$ fixed_subset $G\}$ $(-x) \in$ (fixed_subset G)
= $[\lambda x :$ fixed_subset $G]$ pair (neg_is_fixed x) (neg_closed x.ev.snd)

▷ Prove fixed_forms_subgroup : subgroup_axioms (fixed_subset G)
= pair$_3$ P_ze P_plus P_neg

▷ Define fixed_subgroup = (fixed_subset G, fixed_forms_subgroup : subgroup F)
: subgroup F

▷ Discharge
P_neg, P_plus, P_ze, neg_is_fixed, Q_4, Q_3, plus_is_fixed, Q_2, Q_1, ze_is_fixed, gg

▷ Introduce x, y : fixed_subgroup; $g : G$ and let gg = $G._{\mathbf{2}}\ g$: Aut L

▷ Prove subresult un_is_fixed : g.fixes $\mathbf{1}$
= gg.on_un

▷ Prove subresult Q_5
: (gg $(x$.ev.snd $\times\ y$.ev.snd$)$) = ((gg x.ev.snd) $\times$ (gg y.ev.snd))
= gg.on_times x.ev.snd y.ev.snd

▷ Prove subresult Q_6 **:** $((g\ x) \times (g\ y)) = (x \times y)$
= times_resp $(x.\mathsf{ev.fst}\ g)$ $(y.\mathsf{ev.fst}\ g)$

▷ Prove subresult times_is_fixed **:** $g.\mathsf{fixes}\ (x \times y)$
= $\mathsf{Q}_5.\mathsf{tran}\ \mathsf{Q}_6$

▷ Prove subresult Q_7 **:** $(\mathsf{gg}\ ({}^1\!/x.\mathsf{ev.snd})) = ({}^1\!/(\mathsf{gg}\ x.\mathsf{ev.snd}))$
= gg.on_recip $x.\mathsf{ev.snd}$

▷ Prove subresult Q_8 **:** $({}^1\!/(g\ x)) = ({}^1\!/x)$
= recip_resp $(x.\mathsf{ev.fst}\ g)$

▷ Prove subresult recip_is_fixed **:** $g.\mathsf{fixes}\ ({}^1\!/x)$
= $\mathsf{Q}_7.\mathsf{tran}\ \mathsf{Q}_8$

▷ Discharge g, y, x

▷ Prove subresult P_un **:** $\mathbf{1} \in$ fixed_subgroup
= pair un_is_fixed $(\mathsf{un\_closed}|F|L)$

▷ Prove subresult P_times **:** $\{\forall x, y : \mathsf{fixed\_subgroup}\}\ (x \times y) \in \mathsf{fixed\_subgroup}$
= $[\lambda x, y : \mathsf{fixed\_subgroup}]$ pair (times_is_fixed $x\ y$) (times_closed $x.\mathsf{ev.snd}\ y.\mathsf{ev.snd}$)

▷ Prove subresult P_recip **:** $\{\forall x : \mathsf{fixed\_subgroup}\}\ ({}^1\!/x) \in \mathsf{fixed\_subgroup}$
= $[\lambda x : \mathsf{fixed\_subgroup}]$ pair (recip_is_fixed x) (recip_closed $x.\mathsf{ev.snd}$)

▷ Prove fixed_forms_subfield **:** subfield_axioms fixed_subgroup
= pair_3 P_un P_times P_recip

▷ Define fixed_subfield $=$ (fixed_subgroup, fixed_forms_subfield : subfield F)
: subfield F

▷ Prove fix_forms_subfield **:** fixed_subfield $\subseteq L$
= $[\lambda x : \mathsf{fixed\_subfield}]\ x.\mathsf{ev.snd}$

▷ Define fix $=$ (fixed_subfield, fix_forms_subfield : subfield_of L) **:** subfield_of L

▷ Discharge P_recip, P_times, P_un, recip_is_fixed, Q_8, Q_7, times_is_fixed, Q_6, Q_5, un_is_fixed, gg, G

▷ Discharge L

▷ Allow $^\triangledown$ to be written postfix

▷ Allow $^\triangle$ to be written postfix

▷ Define $^\triangledown = [\lambda L \,|\, \mathsf{subfield}\ F]\ \mathsf{fix}|L$
: $\{\Pi L \,|\, \mathsf{subfield}\ F\}\ (\mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)) \to \mathsf{subfield\_of}\ L$

▷ Define $^{\Delta} = [\lambda L \,|\, \mathsf{subfield}\ F]\ [\lambda K : \mathsf{subfield\_of}\ L]\ \mathsf{aut}\ L\ K$

$: \{\Pi L \,|\, \mathsf{subfield}\ F\}\ (\mathsf{subfield\_of}\ L) \to \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)$

B.8.1.4 The Galois connection between fixing subgroups and fixed subfields

▷ Introduce L | $\mathsf{subfield}\ F$

▷ Construct by refinement GC_1
$: \{\forall U, G \,|\, \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)\}\ (U \subseteq G) \to G^{\nabla} \subseteq U^{\nabla}$
($^{\nabla}$, ev, $\in$, fixes, snd, fst, pair, perm, $\subseteq$, Aut, subgroup_of)

▷ Construct by refinement $\mathsf{GC}_1\mathsf{eq}$
$: \{\forall U, G \,|\, \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)\}\ (U = G) \to U^{\nabla} = G^{\nabla}$
(perm, $\subseteq$, fst, GC_1, snd, $^{\nabla}$, pair, $=$, Aut, subgroup_of)

▷ Construct by refinement GC_2 $: \{\forall J, K \,|\, \mathsf{subfield\_of}\ L\}\ (J \subseteq K) \to K^{\Delta} \subseteq J^{\Delta}$
($^{\Delta}$, perm, ev, Aut, $\in$, fixing_group, snd, fst, pair, $\subseteq$, subfield_of)

▷ Construct by refinement $\mathsf{GC}_2\mathsf{eq}$
$: \{\forall J, K \,|\, \mathsf{subfield\_of}\ L\}\ (J = K) \to J^{\Delta} = K^{\Delta}$
($\subseteq$, fst, GC_2, snd, $^{\Delta}$, perm, pair, $=$, subfield_of)

▷ Construct by refinement GC_3 $: \{\forall G : \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)\}\ G \subseteq G^{\nabla\Delta}$
($^{\nabla}$, ev, $\in$, fixes, fst, Aut, perm, fixing_group, pair, subgroup_of)

▷ Construct by refinement GC_4 $: \{\forall K : \mathsf{subfield\_of}\ L\}\ K \subseteq K^{\Delta\nabla}$
($^{\Delta}$, perm, ev, Aut, $\in$, fixing_group, fst, fixes, pair, subfield_of)

▷ Prove GC_5 $: \{\forall G : \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)\}\ G^{\nabla} = G^{\nabla\Delta\nabla}$
$= [\lambda G : \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)]\ \mathsf{pair}\ (\mathsf{GC}_4\ G^{\nabla})\ (\mathsf{GC}_1\ (\mathsf{GC}_3\ G))$

▷ Prove GC_6 $: \{\forall K : \mathsf{subfield\_of}\ L\}\ K^{\Delta} = K^{\Delta\nabla\Delta}$
$= [\lambda K : \mathsf{subfield\_of}\ L]\ \mathsf{pair}\ (\mathsf{GC}_3\ K^{\Delta})\ (\mathsf{GC}_2\ (\mathsf{GC}_4\ K))$

▷ Discharge L

B.8.1.5 Results concerning subfield morphisms

▷ Introduce K : $\mathsf{subfield}\ F$; J_1, J_2 | $\mathsf{subfield}\ F$; suppose $H_1 : J_1 \subseteq J_2$ and introduce $H_2 : J_2 \subseteq J_1$

▷ Prove subresult $\mathsf{ALG_2s} : (K\text{-}\mathsf{fix\_hom}\ J_1) \subseteq (K\text{-}\mathsf{fix\_hom}\ J_2)$
$= [\lambda g : K\text{-}\mathsf{fix\_hom}\ J_1]$
$\mathsf{pair}\ (\mathsf{FROM_1s}\ H_1\ g.\mathsf{ev.fst})\ (\mathsf{pair_3}\ ([\lambda x, y : J_2]\ g.\mathsf{ev.snd.fst_3}\ (H_2\ x)\ (H_2\ y))$
$([\lambda x, y : J_2]\ g.\mathsf{ev.snd.snd_3}\ (H_2\ x)\ (H_2\ y))\ g.\mathsf{ev.snd.thd_3})$

▷ Discharge H_2, H_1, J_2, J_1

▷ Prove $\mathsf{ALG_2}$
$: \{\forall J_1, J_2 \mid \mathsf{subfield}\ F\}\ (J_1 = J_2) \to (K\text{-}\mathsf{fix\_hom}\ J_1) = (K\text{-}\mathsf{fix\_hom}\ J_2)$
$= [\lambda J_1, J_2 \mid \mathsf{subfield}\ F]\ [\lambda Q : J_1 = J_2]$
$\mathsf{pair}\ (\mathsf{ALG_2s}\ Q.\mathsf{fst}\ Q.\mathsf{snd})\ (\mathsf{ALG_2s}\ Q.\mathsf{snd}\ Q.\mathsf{fst})$

▷ Discharge $\mathsf{ALG_2s}, K$

▷ Introduce $K_1, K_2 \mid \mathsf{subfield}\ F$

▷ Prove $\mathsf{ALG_1s}$
$: (K_1 \subseteq K_2) \to \{\forall J : \mathsf{dsubfield}\ F\}\ (K_2\text{-}\mathsf{fix\_hom}\ J) \subseteq (K_1\text{-}\mathsf{fix\_hom}\ J)$
$= [\lambda H : K_1 \subseteq K_2]\ [\lambda J : \mathsf{dsubfield}\ F]\ [\lambda g : K_2\text{-}\mathsf{fix\_hom}\ J]$
$\mathsf{pair}\ g.\mathsf{ev.fst}\ (\mathsf{pair_3}\ g.\mathsf{ev.snd.fst_3}\ g.\mathsf{ev.snd.snd_3}\ ([\lambda x : K_1]\ g.\mathsf{ev.snd.thd_3}\ (H\ x)))$

▷ Discharge but keep K_2, K_1

▷ Prove $\mathsf{ALG_1}$
$: (K_1 = K_2) \to \{\forall J : \mathsf{dsubfield}\ F\}\ (K_1\text{-}\mathsf{fix\_hom}\ J) = (K_2\text{-}\mathsf{fix\_hom}\ J)$
$= [\lambda Q : K_1 = K_2]\ [\lambda J : \mathsf{dsubfield}\ F]\ \mathsf{pair}\ (\mathsf{ALG_1s}\ Q.\mathsf{snd}\ J)\ (\mathsf{ALG_1s}\ Q.\mathsf{fst}\ J)$

▷ Discharge K_2, K_1

▷ Introduce $L \mid \mathsf{subfield}\ F$

▷ Prove $\mathsf{AUTc} : \{\forall g : \mathsf{Aut}\ L\}\ \{\forall x : L\}\ (g\ x) \in L$
$= [\lambda g : \mathsf{Aut}\ L]\ [\lambda x : L]\ \mathsf{PERMc}\ g\ x$

▷ Introduce $G : \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)$

▷ Introduce $J : \mathsf{dsubfield\_of}\ L$ and suppose $H : G^\nabla \subseteq J$

▷ Introduce $g : G{\restriction}J$ and let $\mathsf{g'} = (g.\mathsf{ev})._{\mathbf{1}} : G$

▷ Prove subresult $\mathsf{P_1} : (\mathsf{g'.rep}{\restriction}J) \in (\mathsf{map\_from}\ J)$
$= \mathsf{REST_5}\ J\ \mathsf{g'.rep}$

▷ Introduce $x, y : J$ and $z : G^\nabla$

▷ Prove subresult $\mathsf{Q_1}$
$: (\mathsf{g'.rep}{\restriction}J\ (x.\mathsf{plus\_closed}\ y))\ (=|F._{\mathbf{1}._{\mathbf{1}}})\ (\mathsf{g'.rep}\ (x.\mathsf{plus\_closed}\ y))$
$= (\mathsf{REST_1}\ J\ \mathsf{g'.rep}\ (x.\mathsf{plus\_closed}\ y)).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q_2} : ((G._{\mathbf{2}}\ \mathsf{g'}).\mathsf{make}\ ((J._{\mathbf{2}}\ x) + (J._{\mathbf{2}}\ y))) =$
$(((G._{\mathbf{2}}\ \mathsf{g'}).\mathsf{make}\ (J._{\mathbf{2}}\ x)) + ((G._{\mathbf{2}}\ \mathsf{g'}).\mathsf{make}\ (J._{\mathbf{2}}\ y)))$
$= (G._{\mathbf{2}}\ \mathsf{g'}).\mathsf{make.on\_plus}\ (J._{\mathbf{2}}\ x)\ (J._{\mathbf{2}}\ y)$

▷ Prove subresult Q_3_x : $x \in (\mathsf{g'.rep.agree}\ (\mathsf{g'.rep}\restriction J))$
= $\mathsf{REST}_1\ J\ \mathsf{g'.rep}\ x$

▷ Prove subresult Q_3_y : $y \in (\mathsf{g'.rep.agree}\ (\mathsf{g'.rep}\restriction J))$
= $\mathsf{REST}_1\ J\ \mathsf{g'.rep}\ y$

▷ Prove subresult Q_3 : $((\mathsf{g'.rep}\ x) + (\mathsf{g'.rep}\ y)) = ((\mathsf{g'.rep}\restriction J\ x) + (\mathsf{g'.rep}\restriction J\ y))$
= $\mathsf{plus\_resp}\ \mathsf{Q}_3\_\mathsf{x}\ \mathsf{Q}_3\_\mathsf{y}$

▷ Prove subresult P_2
: $(\mathsf{g'.rep}\restriction J\ (x.\mathsf{plus\_closed}\ y))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}})\ ((\mathsf{g'.rep}\restriction J\ x) + (\mathsf{g'.rep}\restriction J\ y))$
= $\mathsf{Q}_1.\mathsf{tran}\ (\mathsf{Q}_2.\mathsf{tran}\ \mathsf{Q}_3)$

▷ Prove subresult Q_4
: $(\mathsf{g'.rep}\restriction J\ (x.\mathsf{times\_closed}\ y))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}})\ (\mathsf{g'.rep}\ (x.\mathsf{times\_closed}\ y))$
= $(\mathsf{REST}_1\ J\ \mathsf{g'.rep}\ (x.\mathsf{times\_closed}\ y)).\mathsf{symm}$

▷ Prove subresult Q_5 : $((G_{\cdot\mathbf{2}}\ \mathsf{g'}).\mathsf{make}\ ((J_{\cdot\mathbf{2}}\ x) \times (J_{\cdot\mathbf{2}}\ y))) =$
$(((G_{\cdot\mathbf{2}}\ \mathsf{g'}).\mathsf{make}\ (J_{\cdot\mathbf{2}}\ x)) \times ((G_{\cdot\mathbf{2}}\ \mathsf{g'}).\mathsf{make}\ (J_{\cdot\mathbf{2}}\ y)))$
= $(G_{\cdot\mathbf{2}}\ \mathsf{g'}).\mathsf{make.on\_times}\ (J_{\cdot\mathbf{2}}\ x)\ (J_{\cdot\mathbf{2}}\ y)$

▷ Prove subresult Q_6 : $((\mathsf{g'.rep}\ x) \times (\mathsf{g'.rep}\ y)) = ((\mathsf{g'.rep}\restriction J\ x) \times (\mathsf{g'.rep}\restriction J\ y))$
= $\mathsf{times\_resp}\ \mathsf{Q}_3\_\mathsf{x}\ \mathsf{Q}_3\_\mathsf{y}$

▷ Prove subresult P_3
: $(\mathsf{g'.rep}\restriction J\ (x.\mathsf{times\_closed}\ y))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}})\ ((\mathsf{g'.rep}\restriction J\ x) \times (\mathsf{g'.rep}\restriction J\ y))$
= $\mathsf{Q}_4.\mathsf{tran}\ (\mathsf{Q}_5.\mathsf{tran}\ \mathsf{Q}_6)$

▷ Prove subresult P_4 : $(\mathsf{g'.rep}\restriction J\ (H\ z))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}})\ z$
= $(\mathsf{REST}_1\ J\ \mathsf{g'.rep}\ (H\ z)).\mathsf{symm.tran}\ (z.\mathsf{ev.fst}\ \mathsf{g'})$

▷ Discharge z, y, x

▷ Prove subresult P_5 : $(\mathsf{g'.rep}\restriction J) \in (\text{-}\mathsf{fix\_hom}\ G^{\triangledown}\ J)$
= $\mathsf{pair}\ \mathsf{P}_1\ (\mathsf{pair}\ (\mathsf{pair}\ \mathsf{P}_2\ \mathsf{P}_3)\ \mathsf{P}_4)$

▷ Prove ALG_3 : $g \in (\text{-}\mathsf{fix\_hom}\ G^{\triangledown}\ J)$
= $\mathsf{eq\_closed}\ (g.\mathsf{ev})_{\cdot\mathbf{2}}.\mathsf{symm}\ \mathsf{P}_5$

▷ Discharge $\mathsf{P}_5, \mathsf{P}_4, \mathsf{P}_3, \mathsf{Q}_6, \mathsf{Q}_5, \mathsf{Q}_4, \mathsf{P}_2, \mathsf{Q}_3, \mathsf{Q}_3\_\mathsf{y}, \mathsf{Q}_3\_\mathsf{x}, \mathsf{Q}_2, \mathsf{Q}_1, \mathsf{P}_1, \mathsf{g'}, g, H, J, G$

▷ Introduce J, K : subfield F

▷ Prove subresult P_1 : $\{\forall g : \mathsf{aut}\ J\ K\}\ g.\mathsf{rep.rep} \in (K\text{-}\mathsf{fix\_hom}\ J)$
= $[\lambda g : \mathsf{aut}\ J\ K]\ \mathsf{pair}\ g.\mathsf{ev.snd.fst}\ (\mathsf{pair}\ g.\mathsf{ev.snd.snd}\ g.\mathsf{ev.fst})$

▷ Prove ALG_4 : $(\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps} \subseteq (K\text{-}\mathsf{fix\_hom}\ J)$
= $(\mathsf{AA}_4\ (\mathsf{aut}\ J\ K)\ (\mathsf{rep\_map}\ (\mathsf{iso}\ F\ F))\ (K\text{-}\mathsf{fix\_hom}\ J)).\mathsf{snd}\ \mathsf{P}_1$

▷ Discharge P_1, K, J, L

B.8.2 Galois groups and subfields

B.8.2.1 Definition of a Galois group

▷ Define is_galois_group $= [\lambda L \,|\, \text{subfield } F]\ [\lambda G : \text{subgroup\_of } (\text{Aut } L)]$

$\wedge\ G.\text{is\_finite } (L.\text{is\_fin\_dim\_over } G^{\triangledown})$

$: \{\Pi L \,|\, \text{subfield } F\}\ (\text{subgroup\_of } (\text{Aut } L)) \to \text{prop}$

▷ Introduce L : subfield F

▷ Define galois_group $= \langle \Sigma G : \text{subgroup\_of } (\text{Aut } L) \rangle\ G.\text{is\_galois\_group} : \text{Type}_1$

▷ Discharge L

▷ Define coercion ungal_group $= [\lambda L \,|\, \text{subfield } F]\ [\lambda G : \text{galois\_group } L]\ G_{.\mathbf{1}}$

$: \{\Pi L \,|\, \text{subfield } F\}\ (\text{galois\_group } L) \to \text{subgroup\_of } (\text{Aut } L)$

▷ Define coercion make_gg $= [\lambda L \,|\, \text{subfield } F]\ [\lambda G \,|\, \text{subgroup\_of } (\text{Aut } L)]$

$[\lambda H : G.\text{is\_galois\_group}]\ (G, H : \text{galois\_group } L) : \{\Pi L \,|\, \text{subfield } F\}$

$\{\Pi G \,|\, \text{subgroup\_of } (\text{Aut } L)\}\ G.\text{is\_galois\_group} \to \text{galois\_group } L$

▷ Introduce L | subfield F and G : galois_group L

▷ Prove gg_finite : $G.\text{is\_finite}$

$= G_{.\mathbf{2}}.\text{fst}$

▷ Prove gg_findim : $L.\text{is\_fin\_dim\_over } G^{\triangledown}$

$= G_{.\mathbf{2}}.\text{snd}$

▷ Discharge G, L

B.8.2.2 Definition of a Galois subfield

▷ Define is_galois_subfield $= [\lambda K, L : \text{subfield } F]\ \langle \exists G : \text{galois\_group } L \rangle\ G^{\triangledown} = K$

$: (\text{subfield } F) \to (\text{subfield } F) \to \text{Type}_1$

▷ Introduce L : subfield F

▷ Define galois_subfield $= \langle \Sigma K : \text{subfield } F \rangle\ K.\text{is\_galois\_subfield } L : \text{Type}_1$

▷ Discharge L

▷ Prove ungal_subfield_subs : $\{\Pi L \,|\, \text{subfield } F\}\ \{\Pi K : \text{galois\_subfield } L\}\ K_{.\mathbf{1}} \subseteq L$

$= [\lambda L \,|\, \text{subfield } F]\ [\lambda K : \text{galois\_subfield } L]\ K_{.\mathbf{2}.\mathbf{2}}.\text{snd.subs\_tran } (K_{.\mathbf{2}.\mathbf{1}}{}^{\triangledown})_{.\mathbf{2}}$

▷ Define coercion ungal_subfield $= [\lambda L \,|\, \text{subfield } F]\ [\lambda K : \text{galois\_subfield } L]$

$(K_{\cdot\mathbf{1}}, \text{ungal\_subfield\_subs } K : \text{subfield\_of } L)$

$: \{\Pi L \,|\, \text{subfield } F\}\ (\text{galois\_subfield } L) \to \text{subfield\_of } L$

▷ Define coercion make_gf $= [\lambda L, K \,|\, \text{subfield } F]\ [\lambda H : K.\text{is\_galois\_subfield } L]$

$(K, H : \text{galois\_subfield } L)$

$: \{\Pi L, K \,|\, \text{subfield } F\}\ (K.\text{is\_galois\_subfield } L) \to \text{galois\_subfield } L$

▷ Introduce L | subfield F and K : galois_subfield L

▷ Define gf_gg $= K_{\cdot\mathbf{2}\cdot\mathbf{1}}$: galois_group L

▷ Prove gf_equal : $\text{gf\_gg}^{\triangledown} = K$
$= K_{\cdot\mathbf{2}\cdot\mathbf{2}}$

▷ Discharge K, L

B.8.2.3 Results concerning being a Galois group

▷ Introduce L | subfield F

▷ Introduce G_1, G_2 | subgroup_of (Aut L) and suppose $Q : G_1 = G_2$

▷ Construct by refinement $\text{GAL}_1 : G_1.\text{is\_galois\_group} \to G_2.\text{is\_galois\_group}$
($^{\triangledown}$, as_space_over_itself, is_fin_dim_over, perm, is_finite, pair, gg_findim, FINDIM_2, GC_1eq, gg_finite, FIN_4, is_galois_group)

▷ Discharge Q, G_2, G_1, L

B.8.3 Some special subfields

B.8.3.1 Introduction of a particular subfield

▷ Introduce L | dsubfield F

B.8.3.2 Two miscellaneous coercions to aid readability

▷ Define coercion apply_quot $= [\lambda S]\ [\lambda G \,|\, \mathsf{subgroup}\ (\mathsf{perm}\ S)]$

$[\lambda \sim \,|\, \mathsf{equiv\_rel}\ (\mathsf{perm}\ S)]\ [\lambda g : G/\!\sim]\ g.\mathsf{rep}.\mathsf{rep}$ **:** $\{\Pi S\}$

$\{\Pi G \,|\, \mathsf{subgroup}\ (\mathsf{perm}\ S)\}\ \{\Pi \sim \,|\, \mathsf{equiv\_rel}\ (\mathsf{perm}\ S)\}\ (G/\!\sim) \to \mathsf{map}\ S\ S$

▷ Define coercion subg_trans $= [\lambda G \,|\, \mathsf{galois\_group}\ L]\ [\lambda U : \mathsf{subgroup\_of}\ G]$

$(U, U_{.\mathbf{2}}.\mathsf{subs\_tran}\ G_{.\mathbf{1}.\mathbf{2}} : \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L))$

: $\{\Pi G \,|\, \mathsf{galois\_group}\ L\}\ (\mathsf{subgroup\_of}\ G) \to \mathsf{subgroup\_of}\ (\mathsf{Aut}\ L)$

B.8.3.3 Definition of a K-subfield

▷ Allow -subfield to be written postfix

▷ Define is_-subfield $= [\lambda J : \mathsf{dsubfield\_of}\ L]\ [\lambda K : \mathsf{subfield\_of}\ L]$

$\bigwedge_3\ (K \subseteq J)\ (J.\mathsf{is\_fin\_dim\_over}\ K)\ (L.\mathsf{is\_fin\_dim\_over}\ J)$

: $(\mathsf{dsubfield\_of}\ L) \to (\mathsf{subfield\_of}\ L) \to \mathsf{prop}$

▷ Define -subfield $= [\lambda K : \mathsf{subfield\_of}\ L]\ \langle \Sigma J : \mathsf{dsubfield\_of}\ L \rangle\ J.\mathsf{is\_}\text{-}\mathsf{subfield}\ K$

: $(\mathsf{subfield\_of}\ L) \to \mathsf{Type}_1$

▷ Introduce K **:** subfield_of L

▷ Define coercion unsubbfield $= [\lambda J : K\text{-}\mathsf{subfield}]\ J_{.\mathbf{1}}$

: $K\text{-}\mathsf{subfield} \to \mathsf{dsubfield\_of}\ L$

▷ Define coercion make_-subfield $= [\lambda J \,|\, \mathsf{dsubfield\_of}\ L]\ [\lambda H : J.\mathsf{is}\_\text{-}\mathsf{subfield}\ K]$

$(J, H : K\text{-}\mathsf{subfield})$ **:** $\{\Pi J \,|\, \mathsf{dsubfield\_of}\ L\}\ (J.\mathsf{is}\_\text{-}\mathsf{subfield}\ K) \to K\text{-}\mathsf{subfield}$

▷ Discharge K

▷ Prove sub_fin_dim_over

: $\{\forall K : \mathsf{subfield\_of}\ L\}\ \{\forall J : K\text{-}\mathsf{subfield}\}\ J.\mathsf{is\_fin\_dim\_over}\ K$

$= [\lambda K : \mathsf{subfield\_of}\ L]\ [\lambda J : K\text{-}\mathsf{subfield}]\ J_{.\mathbf{2}}.\mathsf{snd}_3$

▷ Prove fin_dim_over_sub

: $\{\forall K \,|\, \mathsf{subfield\_of}\ L\}\ \{\forall J : K\text{-}\mathsf{subfield}\}\ L.\mathsf{is\_fin\_dim\_over}\ J$

$= [\lambda K \,|\, \mathsf{subfield\_of}\ L]\ [\lambda J : K\text{-}\mathsf{subfield}]\ J_{.\mathbf{2}}.\mathsf{thd}_3$

▷ Prove subs_sub **:** $\{\forall K \,|\, \mathsf{subfield\_of}\ L\}\ \{\forall J : K\text{-}\mathsf{subfield}\}\ K \subseteq J$

$= [\lambda K \,|\, \mathsf{subfield\_of}\ L]\ [\lambda J : K\text{-}\mathsf{subfield}]\ J_{.\mathbf{2}}.\mathsf{fst}_3$

▷ Prove sub_subs : $\{\forall K \mid \mathsf{subfield\_of}\ L\}\ \{\forall J : K\text{-subfield}\}\ J \subseteq L$
$= [\lambda K \mid \mathsf{subfield\_of}\ L]\ [\lambda J : K\text{-subfield}]\ J_{.\mathbf{1}.\mathbf{2}}$

B.8.3.4 Results concerning K-subfields

▷ Prove SUBF_1 : $\{\Pi K_1, K_2 \mid \mathsf{subfield\_of}\ L\}\ (K_1 = K_2) \to \{\Pi J : K_1\text{-subfield}\}$
$J.\mathsf{is\_}$-subfield K_2
$= [\lambda K_1, K_2 \mid \mathsf{subfield\_of}\ L]\ [\lambda Q : K_1 = K_2]\ [\lambda J : K_1\text{-subfield}]$
$\mathsf{pair}_3\ (Q.\mathsf{snd.subs\_tran}\ J.\mathsf{subs\_sub})\ (\mathsf{FINDIM}_2\ J\ Q\ (\mathsf{sub\_fin\_dim\_over}\ K_1\ J))$
$(\mathsf{fin\_dim\_over\_sub}\ J)$

B.8.3.5 Decideability of a restricted subgroup

▷ Introduce G : subgroup_of (Aut L); let $\mathsf{K} = G^{\triangledown}$: subfield_of L and suppose
H : G.is_finite

▷ Introduce J : K-subfield and let $\mathsf{GbJ} = G{\restriction}J$: subset (map F F)

▷ We want to prove subresult P_1 : GbJ.is_decideable_in (K-fix_hom J)

▷ Introduce g : K-fix_hom J

▷ Let $\mathsf{n} = H_{.\mathbf{1}}$: $\mathbb{N}$

▷ Let $\boldsymbol{g} = H_{.\mathbf{2}.\mathbf{1}}$: $G^{\wedge}\mathsf{n}$

▷ Let $\mathsf{rtJ} = \mathsf{restriction}\ J$: map (map $F_{.\mathbf{1}.\mathbf{1}}$ $F_{.\mathbf{1}.\mathbf{1}}$) (map $F_{.\mathbf{1}.\mathbf{1}}$ $F_{.\mathbf{1}.\mathbf{1}}$)

▷ Prove subresult P_2 : subset_axioms|G $([\lambda g' : G]\ g = g'.\mathsf{rtJ})$
$= [\lambda g_1, g_2 \mid G]\ [\lambda Q g : g_1 = g_2]\ [\lambda Q : g = g_1.\mathsf{rtJ}]\ Q.\mathsf{tran}\ (\mathsf{rtJ.resp}\ Qg)$

▷ Let $\mathsf{A} = ([\lambda g' : G]\ g = g'.\mathsf{rtJ}, \mathsf{P}_2 : \mathsf{subset}\ G)$: subset G

▷ We want to prove subresult P_3 : $(\langle \exists i : \mathsf{n} \rangle\ g = (\boldsymbol{g}_{\backslash} i).\mathsf{rtJ}).\mathsf{or\_not}$

▷ We want to prove subresult P_4 : $\{\forall i : \mathsf{n}\}\ (g = (\boldsymbol{g}_{\backslash} i).\mathsf{rtJ}).\mathsf{or\_not}$

▷ Prove subresult P_{4_3} : subset_axioms|n $([\lambda i : \mathsf{n}]\ g = (\boldsymbol{g}_{\backslash} i).\mathsf{rtJ})$
$= [\lambda i_1, i_2 \mid \mathsf{n}]\ [\lambda Q i : i_1 = i_2]\ [\lambda Q : g = (\boldsymbol{g}_{\backslash} i_1).\mathsf{rtJ}]\ Q.\mathsf{tran}\ (\mathsf{rtJ.resp}\ (\boldsymbol{g}.\mathsf{resp}\ Qi))$

▷ Let $\mathsf{B} = ([\lambda i : \mathsf{n}]\ g = (\boldsymbol{g}_{\backslash} i).\mathsf{rtJ}, \mathsf{P}_{4\_3} : \mathsf{subset}\ \mathsf{n})$: subset n

▷ Introduce i : n

▷ Let $\mathsf{gi} = \boldsymbol{g}_{\backslash} i$: G

▷ We want to prove subresult $\mathsf{P}_{4\text{-}1}$

: $(\{\forall x : F\}\ (x \in J) \to (g\ x) = (\mathsf{gi.rtJ}\ x)).\mathsf{or\_not}$

▷ We want to prove subresult $\mathsf{P}_{4\text{-}2}$

: $(\{\forall x : F\}\ (x \notin J) \to (g\ x) = (\mathsf{gi.rtJ}\ x)).\mathsf{or\_not}$

▷ Prove subresult $\mathsf{Q}_{3\text{-}1}$

: $(J \subseteq (g.\mathsf{agree\ gi})) \to \{\forall x : F\}\ (x \in J) \to (g\ x) = (\mathsf{gi.rtJ}\ x)$

$= [\lambda H_1 : J \subseteq (g.\mathsf{agree\ gi})]\ [\lambda x : F]\ [\lambda H J : x \in J]\ (H_1\ H J).\mathsf{tran}\ (\mathsf{REST}_1\ J\ \mathsf{gi}\ H J)$

▷ Prove subresult $\mathsf{Q}_{3\text{-}2}$

: $(\{\Pi x : F\}\ (x \in J) \to (g\ x) = (\mathsf{gi.rtJ}\ x)) \to J \subseteq (g.\mathsf{agree\ gi})$

$= [\lambda H_2 : \{\Pi x : F\}\ (x \in J) \to (g\ x) = (\mathsf{gi.rtJ}\ x)]\ [\lambda x : J]$
$(H_2\ x\ x.\mathsf{ev}).\mathsf{tran}\ (\mathsf{REST}_1\ J\ \mathsf{gi}\ x.\mathsf{ev}).\mathsf{symm}$

▷ Prove subresult Q_3

: $(J \subseteq (g.\mathsf{agree\ gi})) \leftrightarrow (\{\forall x : F\}\ (x \in J) \to (g\ x) = (\mathsf{gi.rtJ}\ x))$

$= \mathsf{pair}\ \mathsf{Q}_{3\text{-}1}\ \mathsf{Q}_{3\text{-}2}$

▷ Prove subresult $\mathsf{P}_{4\text{-}1\text{-}1}$: $J.\mathsf{has\_fin\_span\_over}\ \mathsf{K}$

$= \mathsf{span\_fin\_dim}\ (\mathsf{sub\_fin\_dim\_over}\ \mathsf{K}\ J)$

▷ Let $\mathsf{P}_{4\text{-}1\text{-}2} = [\lambda x, y : J]\ g.\mathsf{ev.snd.fst.fst}\ x\ y$: $g \in (\mathsf{subgroup\_hom}\ J)$

▷ Let $\mathsf{P}_{4\text{-}1\text{-}3} = [\lambda x, y : J]\ (G_{\cdot\mathbf{2}}\ \mathsf{gi}).\mathsf{snd.fst}\ (J.\mathsf{sub\_subs}\ x)\ (J.\mathsf{sub\_subs}\ y)$

: $\mathsf{gi.rep.rep} \in (\mathsf{subgroup\_hom}\ J)$

▷ We want to prove subresult $\mathsf{P}_{4\text{-}1\text{-}4}$

: $\{\forall v : J \cap (g.\mathsf{agree\ gi})\}\ \{\forall x : \mathsf{K}\}\ (x \times v) \in (g.\mathsf{agree\ gi})$

▷ Introduce v : $J \cap (g.\mathsf{agree\ gi})$; let $\mathsf{Pv}_1 = v.\mathsf{ev.fst}$: $v \in J$ and Pv_2

$= J.\mathsf{sub\_subs}\ \mathsf{Pv}_1$: $v \in L$

▷ Introduce x : K; let $\mathsf{Px}_1 = J.\mathsf{subs\_sub}\ x$: $x \in J$ and $\mathsf{Px}_2 = x.\mathsf{ev.snd}$: $x \in L$

▷ Prove subresult $\mathsf{P}_{4\text{-}1\text{-}4}\mathsf{a}$: $(g\ (x \times v)) = ((g\ x) \times (g\ v))$

$= g.\mathsf{ev.snd.fst.snd}\ \mathsf{Px}_1\ \mathsf{Pv}_1$

▷ Prove subresult $\mathsf{P}_{4\text{-}1\text{-}4}\mathsf{b}_1$: $g.\mathsf{fixes}\ x$

$= g.\mathsf{ev.snd.snd}\ x$

▷ Prove subresult $\mathsf{P}_{4\text{-}1\text{-}4}\mathsf{b}_2$: $x\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (\mathsf{gi}\ x)$

$= (x.\mathsf{ev.fst\ gi}).\mathsf{symm}$

▷ Prove subresult $\mathsf{P}_{4\text{-}1\text{-}4}\mathsf{b}$: $(g\ x)\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}})\ (\mathsf{gi}\ x)$

$= \mathsf{P}_{4\text{-}1\text{-}4}\mathsf{b}_1.\mathsf{tran}\ \mathsf{P}_{4\text{-}1\text{-}4}\mathsf{b}_2$

▷ Prove subresult $\mathsf{P}_{4\text{-}1\text{-}4}\mathsf{c}$: $((g\ x) \times (g\ v)) = ((\mathsf{gi}\ x) \times (\mathsf{gi}\ v))$
= times_resp $\mathsf{P}_{4\text{-}1\text{-}4}$b v.ev.snd

▷ Prove subresult $\mathsf{P}_{4\text{-}1\text{-}4}\mathsf{d}$: $((\mathsf{gi}\ x) \times (\mathsf{gi}\ v)) = (\mathsf{gi}\ (x \times v))$
= $((G.\mathbf{2}\ \mathsf{gi}).\mathsf{snd}.\mathsf{snd}\ \mathsf{Px}_2\ \mathsf{Pv}_2).\mathsf{symm}$

▷ Discharge to prove subresult as claimed $\mathsf{P}_{4\text{-}1\text{-}4}$
: $\{\forall v : J \cap (g.\mathsf{agree}\ \mathsf{gi})\}\ \{\forall x : \mathsf{K}\}\ (x \times v) \in (g.\mathsf{agree}\ \mathsf{gi})$
using
$\mathsf{P}_{4\text{-}1\text{-}4}$a.tran ($\mathsf{P}_{4\text{-}1\text{-}4}$c.tran $\mathsf{P}_{4\text{-}1\text{-}4}$d) : $(x \times v) \in (g.\mathsf{agree}\ \mathsf{gi})$

▷ Discharge x, v

▷ Prove subresult $\mathsf{P}_{4\text{-}1}$' : $(J \subseteq (g.\mathsf{agree}\ \mathsf{gi})).\mathsf{or\_not}$
= $\mathsf{SPAN}_4|F|J|\mathsf{K}|\mathsf{P}_{4\text{-}1\text{-}2}|\mathsf{P}_{4\text{-}1\text{-}3}$ J.subs_sub $\mathsf{P}_{4\text{-}1\text{-}1}$ $\mathsf{P}_{4\text{-}1\text{-}4}$

▷ Prove subresult as claimed $\mathsf{P}_{4\text{-}1}$
: $(\{\forall x : F\}\ (x \in J) \to (g\ x) = (\mathsf{gi.rtJ}\ x)).\mathsf{or\_not}$
= DEC_2 Q_3 $\mathsf{P}_{4\text{-}1}$'

▷ Introduce $x : F$ and $X : x \notin J$

▷ Prove subresult $\mathsf{P}_{4\text{-}2\text{-}1}$: $(x \in J) \to (g\ x) = (\mathsf{gi.rtJ}\ x)$
= $[\lambda ABS : x \in J]$ ex_falso $((g\ x) = (\mathsf{gi.rtJ}\ x))$ $(X\ ABS)$

▷ Prove subresult Q_4 : $(\mathsf{gi.rtJ}\ x) = x$
= REST_2 J gi X

▷ Prove subresult $\mathsf{P}_{4\text{-}2\text{-}2}$: $((g\ x) = x) \to (g\ x)\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (\mathsf{gi.rtJ}\ x)$
= $[\lambda Q : (g\ x) = x]$ Q.tran Q_4.symm

▷ Prove subresult $\mathsf{P}_{4\text{-}2}$' : $(g\ x) = (\mathsf{gi.rtJ}\ x)$
= case $(g.\mathsf{ev.fst}\ x)$ $\mathsf{P}_{4\text{-}2\text{-}1}$ $\mathsf{P}_{4\text{-}2\text{-}2}$

▷ Discharge X, x

▷ Prove subresult as claimed $\mathsf{P}_{4\text{-}2}$
: $(\{\forall x : F\}\ (x \notin J) \to (g\ x) = (\mathsf{gi.rtJ}\ x)).\mathsf{or\_not}$
= or_not_is_true $\mathsf{P}_{4\text{-}2}$'

▷ Discharge to prove subresult as claimed P_4 : $\{\forall i : \mathsf{n}\}\ (g = (\boldsymbol{g}_{\setminus} i).\mathsf{rtJ}).\mathsf{or\_not}$
using
DEC_2 (DEC_1 ($[\lambda x : F]\ (g\ x) = (\mathsf{gi.rtJ}\ x)$) J) (DEC_3 $\mathsf{P}_{4\text{-}1}$ $\mathsf{P}_{4\text{-}2}$)
: $(\{\Pi x : F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}}\}\ [\lambda x' : F]\ (g\ x') = (\mathsf{gi.rtJ}\ x')\ x).\mathsf{or\_not}$

▷ Discharge i

▷ Prove subresult as claimed P_3 : $(\langle \exists i : \mathsf{n} \rangle\ g = (\boldsymbol{g}_{\setminus} i).\mathsf{rtJ}).\mathsf{or\_not}$
= FIN_5 B P_4

▷ Discharge to prove subresult as claimed P_1
: GbJ.is_decideable_in (K-fix_hom J)
using
DEC_2 (FIN_7 H A).iff_symm P_3 : ($\langle\exists x : G\rangle$ $x \in$ A).or_not

▷ Discharge g

▷ Prove subresult P_5 : GbJ $\subseteq$ (K-fix_hom J)
$=$ ALG_3 G J (subs_sub J)

▷ Prove GBJ : GbJ.decideable_subs (K-fix_hom J)
$=$ pair P_1 P_5

▷ Discharge $\mathsf{P}_5, \mathsf{P}_1, \mathsf{P}_3, \mathsf{P}_4, \mathsf{P}_{4\text{–}2}, \mathsf{P}_{4\text{–}2}', \mathsf{P}_{4\text{–}2\text{–}2}, \mathsf{Q}_4, \mathsf{P}_{4\text{–}2\text{–}1}, \mathsf{P}_{4\text{–}1}, \mathsf{P}_{4\text{–}1}', \mathsf{P}_{4\text{–}1\text{–}4},$
$\mathsf{P}_{4\text{–}1\text{–}4}\mathsf{d}, \mathsf{P}_{4\text{–}1\text{–}4}\mathsf{c}, \mathsf{P}_{4\text{–}1\text{–}4}\mathsf{b}, \mathsf{P}_{4\text{–}1\text{–}4}\mathsf{b}_2, \mathsf{P}_{4\text{–}1\text{–}4}\mathsf{b}_1, \mathsf{P}_{4\text{–}1\text{–}4}\mathsf{a}, \mathsf{Px}_2, \mathsf{Px}_1, \mathsf{Pv}_2, \mathsf{Pv}_1,$
$\mathsf{P}_{4\text{–}1\text{–}3}, \mathsf{P}_{4\text{–}1\text{–}2}, \mathsf{P}_{4\text{–}1\text{–}1}, \mathsf{Q}_3, \mathsf{Q}_{3\text{–}2}, \mathsf{Q}_{3\text{–}1}, \mathsf{gi}, \mathsf{B}, \mathsf{P}_{4\text{–}3}, \mathsf{A}, \mathsf{P}_2, \mathsf{rtJ}, \boldsymbol{g}, \mathsf{n}, \mathsf{GbJ}, J, H, \mathsf{K}, G$

B.9 Statement of the fundamental theorem

▷ Introduce G : galois_group L

▷ Let $\mathsf{ft}_1\mathsf{i} = \{\forall U : \mathsf{subgroup\_of}\ G\}$ (U.is_decideable_in G) $\to$
(L.is_fin_dim_over U^∇) $\to$ U.is_galois_group : prop_1

▷ Let $\mathsf{ft}_1\mathsf{ii} = G^\nabla$.is_galois_subfield L : prop_1

▷ Let $\mathsf{ft}_1\mathsf{iii} = G = G^{\nabla\triangle}$: prop

▷ We want to prove FT_1 : $\bigwedge_3$ $\mathsf{ft}_1\mathsf{i}$ $\mathsf{ft}_1\mathsf{ii}$ $\mathsf{ft}_1\mathsf{iii}$

▷ Introduce K : galois_subfield L

▷ Let $\mathsf{ft}_2\mathsf{i} = \{\forall J : K\text{-subfield}\}$ J.is_galois_subfield L : prop_1

▷ Let $\mathsf{ft}_2\mathsf{ii} = K^\triangle$.is_galois_group : prop

▷ Let $\mathsf{ft}_2\mathsf{iii} = K = K^{\triangle\nabla}$: prop

▷ We want to prove FT_2 : $\bigwedge_3$ $\mathsf{ft}_2\mathsf{i}$ $\mathsf{ft}_2\mathsf{ii}$ $\mathsf{ft}_2\mathsf{iii}$

▷ Introduce J : K-subfield

▷ Let $\mathsf{ft}_3\mathsf{A} = K$.is_galois_subfield J : prop_1

▷ Let $\mathsf{ft_3B} = J^{\triangle}.\mathsf{normal\_in}\ K^{\triangle}$: prop

▷ We want to prove $\mathsf{FT_3}$: $(\mathsf{ft_3A} \leftrightarrow \mathsf{ft_3B}) \wedge$

$(\mathsf{ft_3A} \rightarrow \langle \exists \pi : \mathsf{iso}\ (K^{\triangle}/J^{\triangle})\ (\mathsf{aut}\ J\ K)\rangle\ (\{\forall g : K^{\triangle}/J^{\triangle}\}\ (g\restriction J) = (\pi\ g)) \wedge$

$(\{\forall g, h : K^{\triangle}/J^{\triangle}\}\ ((h \circ g)\restriction J) = ((h\restriction J) \circ (g\restriction J))))$

▷ Discharge $\mathsf{ft_3B}, \mathsf{ft_3A}, J, \mathsf{ft_2iii}, \mathsf{ft_2ii}, \mathsf{ft_2i}, K, \mathsf{ft_1iii}, \mathsf{ft_1ii}, \mathsf{ft_1i}, G$

B.10 Proof of the fundamental theorem

B.10.1 Some subresults

B.10.1.1 Lemma 1

B.10.1.1.1 Statement of Lemma 1

▷ Introduce G : subgroup_of (Aut L); J : dsubfield_of L and let $U = G \cap J^{\triangle}$

: subgroup (perm F)

▷ We want to prove $\mathsf{LEMMA_1i}$

: $\{\forall g_1, g_2 : G\}\ (g_1.(\approx\text{-}U)\ g_2) \leftrightarrow ((g_1\restriction J) = (g_2\restriction J))$

▷ We want to prove $\mathsf{LEMMA_1ii}$: $(G\restriction J) \cong (G/U)$

B.10.1.1.2 Proof of the first part

▷ Introduce g_1, g_2 : G

▷ Prove subresult $\mathsf{P_1}$: $({g_1}^{-1} \circ g_2) \in G$
= g_1.inv_closed.o_closed g_2

▷ Prove subresult $\mathsf{Q_{1\_1}}$: $(({g_1}^{-1} \circ g_2) \in U) \rightarrow ({g_1}^{-1} \circ g_2) \in J^{\triangle}$
= $[\lambda H : ({g_1}^{-1} \circ g_2) \in U]\ H$.snd

▷ Prove subresult $\mathsf{Q_{1\_2}}$: $(({g_1}^{-1} \circ g_2) \in J^{\triangle}) \rightarrow ({g_1}^{-1} \circ g_2) \in U$
= $[\lambda H : ({g_1}^{-1} \circ g_2) \in J^{\triangle}]$ $(\mathsf{P_1}, H)$

▷ Prove subresult $\mathsf{Q_1}$: $((({g_1}^{-1} \circ g_2) \in U) \rightarrow ({g_1}^{-1} \circ g_2) \in J^{\triangle}) \wedge$
$((({g_1}^{-1} \circ g_2) \in J^{\triangle}) \rightarrow ({g_1}^{-1} \circ g_2) \in U)$
= pair $\mathsf{Q_{1\_1}}$ $\mathsf{Q_{1\_2}}$

▷ Prove subresult P_2 **:** $({g_1}^{-1} \circ g_2) \in (\mathsf{Aut}\ L)$
$= G_{.\mathbf{2}}\ \mathsf{P}_1$

▷ Prove subresult $\mathsf{Q}_{2\text{-}1}$ **:** $(({g_1}^{-1} \circ g_2) \in J^{\triangle}) \to \{\forall x : J\}\ ({g_1}^{-1} \circ g_2\ x) = x$
$= [\lambda H : ({g_1}^{-1} \circ g_2) \in J^{\triangle}]\ H.\mathsf{fst}$

▷ Prove subresult $\mathsf{Q}_{2\text{-}2}$ **:** $(\{\forall x : J\}\ ({g_1}^{-1} \circ g_2\ x) = x) \to ({g_1}^{-1} \circ g_2) \in J^{\triangle}$
$= [\lambda H : \{\forall x : J\}\ ({g_1}^{-1} \circ g_2\ x) = x]\ (H, \mathsf{P}_2)$

▷ Prove subresult Q_2 **:** $((({g_1}^{-1} \circ g_2) \in J^{\triangle}) \to \{\forall x : J\}\ ({g_1}^{-1} \circ g_2\ x) = x) \wedge$
$((\{\forall x : J\}\ ({g_1}^{-1} \circ g_2\ x) = x) \to ({g_1}^{-1} \circ g_2) \in J^{\triangle})$
$= \mathsf{pair}\ \mathsf{Q}_{2\text{-}1}\ \mathsf{Q}_{2\text{-}2}$

▷ Introduce x **:** J

▷ Prove subresult $\mathsf{Q}_3\mathsf{a}$ **:** $({g_1}^{-1} \circ g_2\ x) = ({g_1}^{-1}\ (g_2\ x))$
$= \mathsf{rewrite\_o}\ {g_1}^{-1}\ g_2\ x$

▷ Prove subresult $\mathsf{Q}_3\mathsf{b}_1$ **:** $(g_1\ ({g_1}^{-1}\ (g_2\ x)))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (g_1 \circ {g_1}^{-1}\ (g_2\ x))$
$= (\mathsf{rewrite\_o}\ g_1\ {g_1}^{-1}\ (g_2\ x)).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q}_3\mathsf{b}_2$ **:** $(((g_1 \circ {g_1}^{-1}))_{\cdot\mathbf{1}\cdot\mathbf{1}}\ (g_2\ x)) = ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ (g_2\ x))$
$= g_1.\mathsf{o\_inv}\ (g_2\ x)$

▷ Prove subresult $\mathsf{Q}_3\mathsf{b}$ **:** $(g_1\ ({g_1}^{-1}\ (g_2\ x))) = (g_2\ x)$
$= \mathsf{Q}_3\mathsf{b}_1.\mathsf{tran}\ \mathsf{Q}_3\mathsf{b}_2$

▷ Prove subresult $\mathsf{Q}_3\mathsf{c}$ **:** $(((g_1^{-1} \circ g_1))_{\cdot\mathbf{1}\cdot\mathbf{1}}\ x) = ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ x)$
$= g_1.\mathsf{inv\_o}\ x$

▷ Discharge x

▷ Prove subresult $\mathsf{Q}_{3\text{-}1}$
: $(\{\forall x : J\}\ ({g_1}^{-1} \circ g_2\ x) = x) \to \{\forall x : J\}\ (g_1\ x) = (g_2\ x)$
$= [\lambda H : \{\forall x : J\}\ ({g_1}^{-1} \circ g_2\ x) = x]\ [\lambda x : J]$
$(g_1.\mathsf{resp}\ ((H\ x).\mathsf{symm.tran}\ (\mathsf{Q}_3\mathsf{a}\ x))).\mathsf{tran}\ (\mathsf{Q}_3\mathsf{b}\ x)$

▷ Prove subresult $\mathsf{Q}_{3\text{-}2}$
: $(\{\forall x : J\}\ (g_1\ x) = (g_2\ x)) \to \{\forall x : J\}\ ({g_1}^{-1} \circ g_2\ x) = x$
$= [\lambda H : \{\forall x : J\}\ (g_1\ x) = (g_2\ x)]\ [\lambda x : J]$
$((\mathsf{Q}_3\mathsf{a}\ x).\mathsf{tran}\ ({g_1}^{-1}.\mathsf{resp}\ (H\ x).\mathsf{symm})).\mathsf{tran}\ (\mathsf{Q}_3\mathsf{c}\ x)$

▷ Prove subresult Q_3
: $((\{\forall x : J\}\ ({g_1}^{-1} \circ g_2\ x) = x) \to \{\forall x : J\}\ (g_1\ x) = (g_2\ x)) \wedge$
$((\{\forall x : J\}\ (g_1\ x) = (g_2\ x)) \to \{\forall x : J\}\ ({g_1}^{-1} \circ g_2\ x) = x)$
$= \mathsf{pair}\ \mathsf{Q}_{3\text{-}1}\ \mathsf{Q}_{3\text{-}2}$

▷ Prove subresult Q_4 **:** $(g_1.(=\restriction J)\ g_2) \leftrightarrow ((g_1 \restriction J) = (g_2 \restriction J))$
$= \mathsf{REST}_4\ J\ g_1\ g_2$

▷ Discharge to prove as claimed $\mathsf{LEMMA}_1\mathsf{i}$
: $\{\forall g_1, g_2 : G\}\ (g_1.(\approx\text{-}U)\ g_2) \leftrightarrow ((g_1 \restriction J) = (g_2 \restriction J))$
using
$(\mathsf{Q}_1.\mathsf{iff\_tran}\ \mathsf{Q}_2).\mathsf{iff\_tran}\ (\mathsf{Q}_3.\mathsf{iff\_tran}\ \mathsf{Q}_4)$
: $((g_1{}^{-1} \circ g_2) \in U) \leftrightarrow ((g_1 \restriction J) = (g_2 \restriction J))$

▷ Discharge g_2, g_1

▷ Discharge $\mathsf{Q}_4, \mathsf{Q}_3, \mathsf{Q}_{3\text{-}2}, \mathsf{Q}_{3\text{-}1}, \mathsf{Q}_3\mathsf{c}, \mathsf{Q}_3\mathsf{b}, \mathsf{Q}_3\mathsf{b}_2, \mathsf{Q}_3\mathsf{b}_1, \mathsf{Q}_3\mathsf{a}, \mathsf{Q}_2, \mathsf{Q}_{2\text{-}2}, \mathsf{Q}_{2\text{-}1}, \mathsf{P}_2, \mathsf{Q}_1, \mathsf{Q}_{1\text{-}2}, \mathsf{Q}_{1\text{-}1}, \mathsf{P}_1$

B.10.1.1.3 Proof of the second part

▷ Let $\mathsf{GbJ} = G \restriction J$: subset (map F F)

▷ Let $\mathsf{GqU} = G/U$: subset $((\mathsf{perm}\ F)/U)$

▷ Define $\mathsf{theta\_fun} = [\lambda g : \mathsf{GqU}]\ ((g.\mathsf{ev}).\mathbf{1} \restriction J, (g.\mathsf{ev}).\mathbf{1}, \mathsf{refl}\ ((g.\mathsf{ev}).\mathbf{1} \restriction J) : \mathsf{GbJ})$
: GqU → GbJ

▷ Introduce $g_1, g_2 \mid \mathsf{GqU}$ and suppose $Q : g_1 = g_2$

▷ Prove subresult $\mathsf{Q}_{1\text{-}1} : g_2.(((\approx\text{-}U)).\mathbf{1})\ (g_2.\mathsf{ev}).\mathbf{1}$
$= ((\approx\text{-}U)).\mathbf{2}.\mathsf{snd}_3\ (g_2.\mathsf{ev}).\mathbf{2}$

▷ Prove subresult $\mathsf{Q}_1 : (g_1.\mathsf{ev}).\mathbf{1}.(((\approx\text{-}U)).\mathbf{1})\ (g_2.\mathsf{ev}).\mathbf{1}$
$= (g_1.\mathsf{ev}).\mathbf{2}.(((\approx\text{-}U)).\mathbf{2}.\mathsf{thd}_3)\ (Q.(((\approx\text{-}U)).\mathbf{2}.\mathsf{thd}_3)\ \mathsf{Q}_{1\text{-}1})$

▷ Prove $\mathsf{theta\_forms\_map} : (\mathsf{theta\_fun}\ g_1) = (\mathsf{theta\_fun}\ g_2)$
$= (\mathsf{LEMMA}_1\mathsf{i}\ (g_1.\mathsf{ev}).\mathbf{1}\ (g_2.\mathsf{ev}).\mathbf{1}).\mathsf{fst}\ \mathsf{Q}_1$

▷ Discharge $\mathsf{Q}_1, \mathsf{Q}_{1\text{-}1}, Q, g_2, g_1$

▷ Define $\mathsf{theta\_map} = (\mathsf{theta\_fun}, \mathsf{theta\_forms\_map} : \mathsf{map\ GqU\ GbJ})$
: map GqU GbJ

▷ Define $\mathsf{theta\_inv\_fun} = [\lambda g : \mathsf{GbJ}]$
$((g.\mathsf{ev}).\mathbf{1}.\mathsf{rep}, (g.\mathsf{ev}).\mathbf{1}, ((\approx\text{-}U)).\mathbf{2}.\mathsf{fst}_3\ (\mathsf{refl}\ (g.\mathsf{ev}).\mathbf{1}.\mathsf{rep}) : \mathsf{GqU}) : \mathsf{GbJ} \to \mathsf{GqU}$

▷ Introduce $g_1, g_2 \mid \mathsf{GbJ}$ and suppose $Q : g_1 = g_2$

▷ Prove subresult $\mathsf{Q}_2 : (\mathsf{restriction\_perm}\ J\ (g_1.\mathsf{ev}).\mathbf{1})\ (=|(\mathsf{map}\ F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}}\ F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}}))$
$(\mathsf{restriction\_perm}\ J\ (g_2.\mathsf{ev}).\mathbf{1})$
$= ((g_1.\mathsf{ev}).\mathbf{2}.\mathsf{symm.tran}\ Q).\mathsf{tran}\ (g_2.\mathsf{ev}).\mathbf{2}$

▷ Prove $\mathsf{theta\_inv\_forms\_map} : (\mathsf{theta\_inv\_fun}\ g_1) = (\mathsf{theta\_inv\_fun}\ g_2)$
$= (\mathsf{LEMMA}_1\mathsf{i}\ (g_1.\mathsf{ev}).\mathbf{1}\ (g_2.\mathsf{ev}).\mathbf{1}).\mathsf{snd}\ \mathsf{Q}_2$

▷ Discharge $\mathsf{Q}_2, Q, g_2, g_1$

▷ Define $\mathsf{theta\_inv} = (\mathsf{theta\_inv\_fun}, \mathsf{theta\_inv\_forms\_map} : \mathsf{map\ GbJ\ GqU})$

$: \mathsf{map\ GbJ\ GqU}$

▷ Prove subresult $\mathsf{P}_1 : (\mathsf{theta\_inv} \circ \mathsf{theta\_map}) = \mathsf{identity}$

$= [\lambda g : \mathsf{GqU}]\ (g.\mathsf{ev}).\mathbf{2}$

▷ Prove subresult $\mathsf{P}_2 : (\mathsf{theta\_map} \circ \mathsf{theta\_inv}) = \mathsf{identity}$

$= [\lambda g : \mathsf{GbJ}]\ (g.\mathsf{ev}).\mathbf{2}.\mathsf{symm}$

▷ Prove (but keep unfrozen) $\mathsf{theta\_forms\_iso} : \mathsf{theta\_map} \in (\mathsf{iso\ GqU\ GbJ})$

$= (\mathsf{theta\_inv}, \mathsf{pair}\ \mathsf{P}_1\ \mathsf{P}_2 : \mathsf{theta\_map} \in (\mathsf{iso\ GqU\ GbJ}))$

▷ Define $\theta = (\mathsf{theta\_map}, \mathsf{theta\_forms\_iso} : \mathsf{iso\ GqU\ GbJ}) : \mathsf{iso\ GqU\ GbJ}$

▷ Prove as claimed $\mathsf{LEMMA}_1\mathsf{ii} : (G{\restriction}J) \cong (G/U)$

$= \theta$

▷ Discharge $\mathsf{P}_2, \mathsf{P}_1, \mathsf{GqU}, \mathsf{GbJ}, U, J, G$

B.10.1.2 Theorem A

▷ Introduce $G : \mathsf{galois\_group}\ L$ and let $K = G^{\nabla} : \mathsf{subfield\_of}\ L$

The proofs of the first two parts have not been formalised in the case-study due to a lack of time.

B.10.1.2.1 Statement of Theorem A

▷ Assume without proof $\mathsf{THM\_A}_1$

$: \{\forall J : K\text{-subfield}\}\ (K\text{-fix\_hom}\ J) \preccurlyeq (\mathsf{sub\_fin\_dim\_over}\ K\ J).\mathsf{basis}$

▷ Assume without proof $\mathsf{THM\_A}_2$

$: \{\forall U : \mathsf{subgroup\_of}\ G\}\ \langle \Sigma H : U^{\nabla}.\mathsf{is\_fin\_dim\_over}\ K \rangle\ (G/U) \cong (\mathsf{basis}\ H)$

▷ We want to prove $\mathsf{THM\_A}_3$

$: \langle \exists n : \mathbb{N} \rangle\ \langle \exists \boldsymbol{w} : F^{\wedge} n \rangle\ (L.\mathsf{has\_basis\_over}\ K\ \boldsymbol{w}) \wedge (G \cong \boldsymbol{w})$

B.10.1.2.2 Proof of the corollary

▷ Define $\mathsf{trivial\_subset} = \mathsf{singleton}\ (\mathsf{id}|(\mathsf{perm}\ F)) : \mathsf{subset}\ (\mathsf{perm}\ F)$

▷ Construct by refinement trivial_forms_subgroup
: subgroup_axioms trivial_subset
(trivial_subset, perm, ev, id, symm, o_id, ∘, tran, GROUP_3, $^{-1}$, =, snd, o_resp, id_o, refl, ∈, pair_3)

▷ Define trivial_subgroup
= (trivial_subset, trivial_forms_subgroup : subgroup (perm F))
: subgroup (perm F)

▷ Construct by refinement trivial_forms_subgroup_of : trivial_subgroup $\subseteq G$
(perm, id_closed, trivial_subgroup, ev, id, eq_closed)

▷ Define $\{\mathsf{id}\}$ = (trivial_subgroup, trivial_forms_subgroup_of : subgroup_of G)
: subgroup_of G

▷ Introduce g : perm F

▷ Prove subresult P_0 : $\langle \Sigma H : \{\mathsf{id}\}^\triangledown.\mathsf{is\_fin\_dim\_over}\ K\rangle\ (G/\{\mathsf{id}\}) \cong (\mathsf{basis}\ H)$
= $\mathsf{THM\_A}_2$ $\{\mathsf{id}\}$

▷ Discharge g

▷ Locally construct by refinement P_1 : $\{\mathsf{id}\}^\triangledown = L$
(ev, $\{\mathsf{id}\}$, perm, id, symm, ∈, fixes, pair, $^\triangledown$, snd, ⊆)

▷ Let $H = \mathsf{P}_{0\cdot\mathbf{1}}$: $\{\mathsf{id}\}^\triangledown.\mathsf{is\_fin\_dim\_over}\ K$

▷ Let $\mathsf{P}_2 = \mathsf{FINDIM}_1\mathsf{b}|F|F\ \mathsf{P}_1\ K\ \mathsf{P}_{0\cdot\mathbf{1}\cdot\mathbf{2}\cdot\mathbf{2}}$: $L.\mathsf{has\_basis\_over}\ K\ \mathsf{P}_{0\cdot\mathbf{1}\cdot\mathbf{2}\cdot\mathbf{1}}$

▷ Locally construct by refinement P_3 : is_same_relation $(\approx\text{-}\{\mathsf{id}\}) =$
(perm, GROUP_2, =, id, $^{-1}$, ∘, fst, symm, =, snd, $\{\mathsf{id}\}$, $\approx$-, is_subrelation, pair)

▷ Locally construct by refinement P_4 : $G \cong (G/\{\mathsf{id}\})$
(P_3, perm, =, $\{\mathsf{id}\}$, quot_iso, quot_triv_iso, /, /, eqsize_tran)

▷ Locally construct by refinement $\mathsf{THM_A}_3$_proof
: $\langle \exists n : \mathbb{N}\rangle\ \langle \exists \boldsymbol{w} : F^\wedge n\rangle\ (L.\mathsf{has\_basis\_over}\ K\ \boldsymbol{w}) \wedge (G \cong \boldsymbol{w})$
(P_0, P_4, K, $\{\mathsf{id}\}$, $^\triangledown$, as_space_over_itself, basis, perm, /, /, eqsize_tran, P_2, ≅, has_basis_over, pair, ∧, $^\wedge$, $\mathbb{N}$)

▷ Prove as claimed $\mathsf{THM\_A}_3$
: $\langle \exists n : \mathbb{N}\rangle\ \langle \exists \boldsymbol{w} : F^\wedge n\rangle\ (L.\mathsf{has\_basis\_over}\ K\ \boldsymbol{w}) \wedge (G \cong \boldsymbol{w})$
= $\mathsf{THM_A}_3$_proof

▷ Discharge $\mathsf{THM_A}_3$_proof, $\mathsf{P}_4, \mathsf{P}_3, \mathsf{P}_2, H, \mathsf{P}_1, \mathsf{P}_0$

▷ Discharge but keep K, G

B.10.1.3 Theorem B

▷ Introduce J : K-subfield

▷ Prove (but keep unfrozen) subresult P_4 : J.is_fin_dim_over K
= sub_fin_dim_over K J

▷ Let $\boldsymbol{w}$ = basis P_4 : $J^{\wedge}$(dim P_4)

▷ Prove subresult W : J.has_basis_over K $\boldsymbol{w}$
= $\mathsf{P}_{4 \cdot \mathbf{2} \cdot \mathbf{2}}$

▷ Let $G{\restriction}J = G{\restriction}J$: subset (map F F)

B.10.1.3.1 Statement of Theorem B

▷ We want to prove $\mathsf{THM\_B}_0$: $G{\restriction}J$.is_finite

▷ We want to prove $\mathsf{THM\_B}_1$: $G{\restriction}J = (K$-fix_hom $J)$

▷ We want to prove $\mathsf{THM\_B}_2$: $G{\restriction}J \cong \boldsymbol{w}$

▷ We want to prove $\mathsf{THM\_B}_3$: $J = J^{\Delta\nabla}$

▷ We want to prove $\mathsf{THM\_B}_4$: $(G{\restriction}L) = (K$-fix_hom $L)$

B.10.1.3.2 Preliminaries

▷ Prove subresult P_1 : $(G \cap J^{\Delta}) \subseteq G$
= $[\lambda g : G \cap J^{\Delta}]$ g.ev.fst

▷ Let $U = (G \cap J^{\Delta}, \mathsf{P}_1$: subgroup_of $G)$: subgroup_of G

▷ Prove subresult P_2 : $U \subseteq J^{\Delta}$
= $[\lambda g : U]$ g.ev.snd

▷ Let $G/U = G/U$: subset $((\mathsf{perm}\ F)/U)$

▷ Let K-fix_hom $J = K$-fix_hom J : subset (map F F)

B.10.1.3.3 A cycle of inequalities

▷ Prove subresult Q_1 : $G/U \cong G{\restriction}J$
= (LEMMA$_1$ii|G J).eqsize_symm

▷ Prove (but keep unfrozen) subresult H : U^{∇}.is_fin_dim_over K
= ((THM_A$_2$ G U)).$\mathbf{1}$

▷ Let $\boldsymbol{v} = \mathsf{basis}\ H : U^{\nabla\wedge}(\mathsf{dim}\ H)$

▷ Prove subresult $\mathsf{Q}_2 : K\text{-}\mathsf{fix\_hom}\ J \preccurlyeq \boldsymbol{w}$
$= \mathsf{THM\_A}_1\ G\ J$

▷ Prove subresult $\mathsf{P}_5 : K\text{-}\mathsf{fix\_hom}\ J.\mathsf{is\_finite}$
$= \mathsf{SMALLER}_0\ \mathsf{Q}_2$

▷ Prove subresult $\mathsf{P}_6 : G\restriction J.\mathsf{decideable\_subs}\ K\text{-}\mathsf{fix\_hom}\ J$
$= \mathsf{GBJ}\ G\ G.\mathsf{gg\_finite}\ J$

▷ Prove subresult $\mathsf{Q}_3 : G\restriction J \preccurlyeq K\text{-}\mathsf{fix\_hom}\ J$
$= \mathsf{FIN}_1\ \mathsf{P}_5\ \mathsf{P}_6$

▷ We want to prove subresult $\mathsf{P}_7 : (\boldsymbol{w}\text{-}\mathsf{span\_over}\ K) \subseteq J$

▷ Introduce $x : K$ and $v : J$

▷ Prove subresult $\mathsf{P}_{7\text{-}1} : x \in J$
$= \mathsf{subs\_sub}\ J\ x$

▷ Prove subresult $\mathsf{P}_{7\text{-}2} : (x \times v) \in J$
$= \mathsf{times\_closed}\ \mathsf{P}_{7\text{-}1}\ v$

▷ Discharge v, x

▷ Prove subresult as claimed $\mathsf{P}_7 : (\boldsymbol{w}\text{-}\mathsf{span\_over}\ K) \subseteq J$
$= \mathsf{SPAN}_3|F|F\ \boldsymbol{w}\ \mathsf{P}_{7\text{-}2}$

▷ We want to prove subresult $\mathsf{P}_8 : (\boldsymbol{w}\text{-}\mathsf{span\_over}\ K) \subseteq (\boldsymbol{v}\text{-}\mathsf{span\_over}\ K)$

▷ Prove subresult $\mathsf{P}_{8\text{-}1} : J \subseteq U^{\nabla}$
$= (\mathsf{GC}_4\ (\mathsf{sub\_subs}\ J)).\mathsf{subs\_tran}\ (\mathsf{GC}_1|L|U\ \mathsf{P}_2)$

▷ Prove subresult $\mathsf{P}_{8\text{-}2} : U^{\nabla} \subseteq (\boldsymbol{v}\text{-}\mathsf{span\_over}\ K)$
$= H.\mathsf{spanning.fst}$

▷ Prove subresult as claimed $\mathsf{P}_8 : (\boldsymbol{w}\text{-}\mathsf{span\_over}\ K) \subseteq (\boldsymbol{v}\text{-}\mathsf{span\_over}\ K)$
$= \mathsf{P}_7.\mathsf{subs\_tran}\ (\mathsf{P}_{8\text{-}1}.\mathsf{subs\_tran}\ \mathsf{P}_{8\text{-}2})$

▷ Prove subresult $\mathsf{Q}_4 : \boldsymbol{w} \preccurlyeq \boldsymbol{v}$
$= \mathsf{SPAN}_5|F|F.\mathsf{as\_space\_over\_itself}\ \mathsf{P}_4.\mathsf{independence}\ H.\mathsf{independence}\ \mathsf{P}_8$

▷ Prove subresult $\mathsf{Q}_5 : \boldsymbol{v} \cong G/U$
$= ((\mathsf{THM\_A}_2\ G\ U)).\mathbf{2}.\mathsf{eqsize\_symm}$

▷ Prove subresult $\mathsf{a} : G/U \cong G{\restriction}J$

$= \mathsf{Q}_1$;

$\mathsf{b} : G{\restriction}J \preccurlyeq K\text{-}\mathsf{fix\_hom}\ J$

$= \mathsf{Q}_3$;

$\mathsf{c} : K\text{-}\mathsf{fix\_hom}\ J \preccurlyeq \boldsymbol{w}$

$= \mathsf{Q}_2$;

$\mathsf{d} : \boldsymbol{w} \preccurlyeq \boldsymbol{v}$

$= \mathsf{Q}_4$

and $\mathsf{e} : \boldsymbol{v} \cong G/U$

$= \mathsf{Q}_5$

▷ Prove subresult $\mathsf{d}_1 : (\boldsymbol{w}\text{-}\mathsf{span\_over}\ K) \subseteq J$

$= \mathsf{P}_7$;

$\mathsf{d}_2 : J \subseteq J^{\triangle\,\triangledown}$

$= \mathsf{GC}_4\ (\mathsf{sub\_subs}\ J)$;

$\mathsf{d}_3 : J^{\triangle\,\triangledown} \subseteq U^{\triangledown}$

$= \mathsf{GC}_1|L|U\ \mathsf{P}_2$;

$\mathsf{d}_{123} : (\boldsymbol{w}\text{-}\mathsf{span\_over}\ K) \subseteq (\boldsymbol{v}\text{-}\mathsf{span\_over}\ K)$

$= \mathsf{P}_8$

and $\mathsf{b}_1 : G{\restriction}J.\mathsf{decideable\_subs}\ K\text{-}\mathsf{fix\_hom}\ J$

$= \mathsf{P}_6$

▷ Prove as claimed $\mathsf{THM\_B}_0 : G{\restriction}J.\mathsf{is\_finite}$

$= \mathsf{SMALLER}_0\ \mathsf{b}$

▷ Prove subresult $\mathsf{Q}_6 : \boldsymbol{v} \cong G{\restriction}J$

$= \mathsf{eqsize\_tran}\ \mathsf{Q}_5\ \mathsf{Q}_1$

▷ Prove subresult $\mathsf{P}_9 : \boldsymbol{v}.\mathsf{is\_finite}$

$= \mathsf{EQSIZE}_0\ \mathsf{Q}_6.\mathsf{eqsize\_symm}\ \mathsf{THM\_B}_0$

▷ Prove subresult $\mathsf{Q}_7 : \boldsymbol{v} \preccurlyeq G{\restriction}J$

$= \mathsf{SMALLER}_1\ \mathsf{P}_9\ \mathsf{Q}_6$

▷ Prove subresult $\mathsf{Q}_8 : K\text{-}\mathsf{fix\_hom}\ J \preccurlyeq G{\restriction}J$

$= \mathsf{smaller\_tran}\ \mathsf{Q}_2\ (\mathsf{smaller\_tran}\ \mathsf{Q}_4\ \mathsf{Q}_7)$

▷ Prove subresult $\mathsf{Q}_9 : \boldsymbol{w} \preccurlyeq K\text{-}\mathsf{fix\_hom}\ J$

$= \mathsf{smaller\_tran}\ \mathsf{Q}_4\ (\mathsf{smaller\_tran}\ \mathsf{Q}_7\ \mathsf{Q}_3)$

▷ Prove subresult $\mathsf{Q}_{10} : \boldsymbol{v} \preccurlyeq \boldsymbol{w}$

$= \mathsf{smaller\_tran}\ \mathsf{Q}_7\ (\mathsf{smaller\_tran}\ \mathsf{Q}_3\ \mathsf{Q}_2)$

▷ Prove subresult $\mathsf{Q}_{11} : G{\restriction}J \cong K\text{-}\mathsf{fix\_hom}\ J$

$= \mathsf{smaller\_asymm}\ \mathsf{Q}_8\ \mathsf{Q}_3$

▷ Prove subresult $\mathsf{Q}_{12} : K\text{-}\mathsf{fix\_hom}\ J \cong \boldsymbol{w}$

$= \mathsf{smaller\_asymm}\ \mathsf{Q}_9\ \mathsf{Q}_2$

▷ Prove subresult Q_{13} : $\boldsymbol{w} \cong \boldsymbol{v}$

= smaller_asymm Q_{10} Q_4

▷ Prove subresult b' : $G\restriction J \cong K$-fix_hom J

= Q_{11} ;

c' : K-fix_hom $J \cong \boldsymbol{w}$

= Q_{12}

and d' : $\boldsymbol{w} \cong \boldsymbol{v}$

= Q_{13}

B.10.1.3.4 Conclusion

▷ Prove as claimed $\mathsf{THM\_B}_1$: $G\restriction J = (K$-fix_hom $J)$

= FIN_3 b_1.snd $\mathsf{THM\_B}_0$ b'

▷ Prove as claimed $\mathsf{THM\_B}_2$: $G\restriction J \cong \boldsymbol{w}$

= b'.eqsize_tran c'

▷ Prove subresult $\mathsf{B}_3\mathsf{b}_{-1}$: $J^{\Delta\nabla} \subseteq U^{\nabla}$

= $\mathsf{GC}_1|L|U$ P_2

▷ Prove subresult $\mathsf{B}_3\mathsf{b}_{-2}$: $U^{\nabla} \subseteq (\boldsymbol{v}$-span_over $K)$

= H.spanning.fst

▷ Prove subresult $\mathsf{B}_3\mathsf{b}_{-3}$: $(\boldsymbol{v}$-span_over $K) \subseteq (\boldsymbol{w}$-span_over $K)$

= $\mathsf{SPAN}_6|F|F$.as_space_over_itself P_4.independence H.independence P_8 Q_{13}

▷ Prove subresult $\mathsf{B}_3\mathsf{b}$: $J^{\Delta\nabla} \subseteq J$

= $(\mathsf{B}_3\mathsf{b}_{-1}$.subs_tran $\mathsf{B}_3\mathsf{b}_{-2})$.subs_tran $(\mathsf{B}_3\mathsf{b}_{-3}$.subs_tran $\mathsf{P}_7)$

▷ Prove as claimed $\mathsf{THM\_B}_3$: $J = J^{\Delta\nabla}$

= pair d_2 $\mathsf{B}_3\mathsf{b}$

▷ Discharge $\mathsf{B}_3\mathsf{b}, \mathsf{B}_3\mathsf{b}_{-3}, \mathsf{B}_3\mathsf{b}_{-2}, \mathsf{B}_3\mathsf{b}_{-1}, \mathsf{d}', \mathsf{c}', \mathsf{b}', \mathsf{Q}_{13}, \mathsf{Q}_{12}, \mathsf{Q}_{11}, \mathsf{Q}_{10}, \mathsf{Q}_9, \mathsf{Q}_8, \mathsf{Q}_7, \mathsf{P}_9,$

$\mathsf{Q}_6, \mathsf{b}_1, \mathsf{d}_{123}, \mathsf{d}_3, \mathsf{d}_2, \mathsf{d}_1, \mathsf{e}, \mathsf{d}, \mathsf{c}, \mathsf{b}, \mathsf{a}, \mathsf{Q}_5, \mathsf{Q}_4, \mathsf{P}_8, \mathsf{P}_{8\text{-}2}, \mathsf{P}_{8\text{-}1}, \mathsf{P}_7, \mathsf{P}_{7\text{-}2}, \mathsf{P}_{7\text{-}1}, \mathsf{Q}_3, \mathsf{P}_6,$

$\mathsf{P}_5, \mathsf{Q}_2, \boldsymbol{v}, H, \mathsf{Q}_1, K$-fix_hom $J, G/U, \mathsf{P}_2, U, \mathsf{P}_1, G\restriction J, \mathsf{W}, \boldsymbol{w}, \mathsf{P}_4, J$

▷ Prove subresult P_1 : $K \subseteq L$

= $K.\mathbf{2}$

▷ Prove subresult P_2 : L.is_fin_dim_over K

= G.gg_findim

▷ Prove subresult P_3 : L.is_fin_dim_over L

= FINDIM_3 L

▷ Prove subresult J : (subs_refl L).is_-subfield K

= pair_3 P_1 P_2 P_3

▷ Prove as claimed $\mathsf{THM\_B_4}$: $(G\restriction L) = (K\text{-}\mathsf{fix\_hom}\ L)$
$= \mathsf{THM\_B_1}\ \mathsf{J}$

▷ Discharge $\mathsf{J}, \mathsf{P_3}, \mathsf{P_2}, \mathsf{P_1}, K, G$

B.10.2 The main proof

B.10.2.1 Proof of first part of the fundamental theorem

▷ Introduce G : $\mathsf{galois\_group}\ L$ and U | $\mathsf{subgroup\_of}\ G$

▷ Suppose H_1 : $U.\mathsf{is\_decideable\_in}\ G$ and H_2 : $L.\mathsf{is\_fin\_dim\_over}\ U^{\triangledown}$

B.10.2.1.1 Proof of $\mathbf{FT_1 i}$

▷ Prove subresult $\mathsf{P_1}$: $U.\mathsf{is\_finite}$
$= \mathsf{SMALLER_0}\ (\mathsf{FIN_1}\ G.\mathsf{gg\_finite}\ (\mathsf{pair}\ H_1\ U.\mathbf{2}))$

▷ Prove $\mathsf{FT_1 i}$: $U.\mathsf{is\_galois\_group}$
$= \mathsf{pair}\ \mathsf{P_1}\ H_2$

▷ Discharge $\mathsf{P_1}, H_2, H_1, U$

B.10.2.1.2 Proof of $\mathbf{FT_1 ii}$

▷ Prove $\mathsf{FT_1 ii}$: $G^{\triangledown}.\mathsf{is\_galois\_subfield}\ L$
$= (G, \mathsf{equal\_subs\_refl}\ G^{\triangledown}\ :\ G^{\triangledown}.\mathsf{is\_galois\_subfield}\ L)$

B.10.2.1.3 Proof of $\mathbf{FT_1 iii}$

▷ We want to prove subresult $\mathsf{P_2}$
: $G^{\triangledown\triangle}.\mathsf{as\_a\_subset\_of\_maps} \subseteq G.\mathsf{as\_a\_subset\_of\_maps}$

▷ Prove subresult $\mathsf{P_2 a}$: $G^{\triangledown\triangle}.\mathsf{as\_a\_subset\_of\_maps} \subseteq (G^{\triangledown}\text{-}\mathsf{fix\_hom}\ L)$
$= \mathsf{ALG_4}\ L\ G^{\triangledown}$

▷ Prove subresult $\mathsf{P_2 b}$: $(G^{\triangledown}\text{-}\mathsf{fix\_hom}\ L) \subseteq (G\restriction L)$
$= (\mathsf{THM\_B_4}\ G).\mathsf{snd}$

▷ Prove subresult $\mathsf{P_2 c'}$: $\{\forall g : G\}\ g = (g\restriction L)$
$= [\lambda g : G]\ (\mathsf{REST_3}\ L\ g).\mathsf{fst}\ (G.\mathbf{1}.\mathbf{2}\ g).\mathsf{fst}$

▷ Locally construct by refinement $\mathsf{P_2c} : (G{\restriction}L) \subseteq G.\mathsf{as\_a\_subset\_of\_maps}$
(map, refl, iso, rep_map, $=$, $\mathsf{P_2c'}$, $\restriction$, as_a_subset_of_maps, eq_closed, restriction_perm, perm, $\mathsf{AA_4}$, apply_across, $\subseteq$, $\in$, snd)

▷ Prove subresult as claimed $\mathsf{P_2}$
: $G^{\nabla\triangle}.\mathsf{as\_a\_subset\_of\_maps} \subseteq G.\mathsf{as\_a\_subset\_of\_maps}$
$=$ $\mathsf{P_2a.subs\_tran}$ ($\mathsf{P_2b.subs\_tran}$ $\mathsf{P_2c}$)

▷ Locally construct by refinement $\mathsf{P_3} : G^{\nabla\triangle} \subseteq G$
($\mathsf{P_2}$, $^\nabla$, $^\triangle$, iso, map, rep_map, $=$, $\mathsf{AA_2s'}$)

▷ Prove $\mathsf{FT_1iii} : G = G^{\nabla\triangle}$
$=$ pair ($\mathsf{GC_3}$ G) $\mathsf{P_3}$

▷ Discharge $\mathsf{P_3}, \mathsf{P_2}, \mathsf{P_2c}, \mathsf{P_2c'}, \mathsf{P_2b}, \mathsf{P_2a}$

▷ Discharge to prove as claimed $\mathsf{FT_1}$
: $\{\forall G : \mathsf{galois\_group}\ L\}\ [\delta\mathsf{ft_1i} = \{\forall U : \mathsf{subgroup\_of}\ G\}$
$(U.\mathsf{is\_decideable\_in}\ G) \to (L.\mathsf{is\_fin\_dim\_over}\ U^\nabla) \to U.\mathsf{is\_galois\_group}]$
$[\delta\mathsf{ft_1ii} = G^\nabla.\mathsf{is\_galois\_subfield}\ L]\ [\delta\mathsf{ft_1iii} = G = G^{\nabla\triangle}]\ \bigwedge_3\ \mathsf{ft_1i}\ \mathsf{ft_1ii}\ \mathsf{ft_1iii}$
using
$\mathsf{pair_3'}\ \mathsf{FT_1i}\ \mathsf{FT_1ii}\ \mathsf{FT_1iii}$
: $\bigwedge_3$ ($\{\forall U \mid \mathsf{subgroup\_of}\ G\}$ $(U.\mathsf{is\_decideable\_in}\ G) \to (L.\mathsf{is\_fin\_dim\_over}\ U^\nabla) \to$
$U.\mathsf{is\_galois\_group}$) $(G^\nabla.\mathsf{is\_galois\_subfield}\ L)$ $(G = G^{\nabla\triangle})$

▷ Discharge G

B.10.2.2 Proof of second part of the fundamental theorem

▷ Introduce K : galois_subfield L

▷ Let $\mathsf{G} = K.\mathsf{gf\_gg}$: galois_group L and $\mathsf{Q} = K.\mathsf{gf\_equal} : \mathsf{G}^\nabla = K$

B.10.2.2.1 Proof of $\mathbf{FT_2i}$

▷ Introduce J : K-subfield

▷ Locally construct by refinement $\mathsf{P_1} : J^\triangle \subseteq \mathsf{G}$
(G, $\mathsf{FT_1iii}$, $^\nabla$, $^\triangle$, perm, $\subseteq$, snd, subs_sub, Q, fst, subs_tran, $\mathsf{GC_2}$)

▷ Let $\mathsf{Jaut} = \mathsf{P_1}$: subgroup_of G

▷ Prove subresult $\mathsf{PJ} : J.\mathsf{is\_}$-subfield G^∇
$=$ $\mathsf{SUBF_1}|K|\mathsf{G}^\nabla$ Q.equal_subs_symm J

▷ Prove subresult $\mathsf{Q'} : J = J^{\triangle\nabla}$
$=$ $\mathsf{THM\_B_3}$ G PJ

▷ We want to prove subresult H_1 **:** $J^{\triangle}$.is_decideable_in G

▷ We want to prove subresult H_2 **:** L.is_fin_dim_over $J^{\triangle\triangledown}$

Hypothesis H_1

▷ Introduce g **:** G

▷ Locally construct by refinement P_2 **:** $(g.(=\restriction J)\ \mathsf{id}) \leftrightarrow (g \in J^{\triangle})$
(Aut, perm, $\in$, subfield_of, fixing_group, fst, G, id, $=\restriction$, pair, $J^{\triangle}$)

▷ Prove subresult Hg **:** $g \in$ (subgroup_hom J)
$=$ $[\lambda x, y : J]$ (G.$\mathbf{1}\cdot\mathbf{2}$ g).snd.fst (J.sub_subs x) (J.sub_subs y)

▷ Prove subresult Hi **:** id $\in$ (subgroup_hom J)
$=$ $[\lambda x, y : J]$ refl $(x + y)$

▷ Let A $=$ Hg.agree Hi **:** subset F

▷ Prove subresult P_3a **:** $\mathsf{G}^{\triangledown} \subseteq J$
$=$ Q.fst.subs_tran (subs_sub J)

▷ Prove subresult P_3b **:** J.has_fin_span_over $\mathsf{G}^{\triangledown}$
$=$ span_fin_dim (FINDIM$_2$ J Q.equal_subs_symm (sub_fin_dim_over K J))

▷ We want to prove subresult P_3c **:** $\{\forall v : J \cap \mathsf{A}\}\ \{\forall x : \mathsf{G}^{\triangledown}\}\ (x \times v) \in \mathsf{A}$

▷ Introduce v **:** $J \cap$ A and x **:** $\mathsf{G}^{\triangledown}$

▷ Locally construct by refinement Q_1 **:** $(g\ (x \times v)) = ((g\ x) \times (g\ v))$
(A, $\cap$, ev, $\in$, fst, sub_subs, rep_map, ap, make, G, $^{\triangledown}$, resp_plus_times, as_a_subset_of_perm, perm, Perm, snd, $\times$, resp_map$_2$, iso, map, rep, $+$)

▷ Locally construct by refinement Q_2 **:** $((g\ x) \times (g\ v)) = (x \times v)$
(A, $\cap$, ev, $\in$, snd, G, $^{\triangledown}$, fixes, fst, times_resp)

▷ Discharge to prove subresult as claimed P_3c
: $\{\forall v : J \cap \mathsf{A}\}\ \{\forall x : \mathsf{G}^{\triangledown}\}\ (x \times v) \in \mathsf{A}$
using
Q_1.tran Q_2 **:** g.fixes $(x \times v)$

▷ Discharge x, v

▷ Prove subresult P_3 **:** $(g.(=\restriction J)\ \mathsf{id})$.or_not
$=$ SPAN$_4$ P_3a P_3b P_3c

▷ Discharge to prove subresult as claimed H_1 **:** $J^{\triangle}$.is_decideable_in G
using
DEC$_2$ P_2 P_3 **:** $(g \in J^{\triangle})$.or_not

▷ Discharge g

Hypothesis H_2

▷ Prove subresult as claimed $\mathsf{H_2}$ **:** L.is_fin_dim_over $J^{\Delta\nabla}$
= FINDIM$_2$ L Q' (fin_dim_over_sub J)

▷ Prove subresult $\mathsf{P_4}$ **:** J^{Δ}.is_galois_group
= FT$_1$i G|J^{Δ} H$_1$ H$_2$

▷ Construct by refinement $\mathsf{FT_2i}$ **:** J.is_galois_subfield L
(Q', $^{\Delta}$, $^{\nabla}$, equal_subs_symm, perm, equal_subs_refl, P$_4$, GC$_1$eq, equal_subs_tran,
=, galois_group)

▷ Discharge $\mathsf{P_4}, \mathsf{H_2}, \mathsf{H_1}, \mathsf{P_3}, \mathsf{P_3c}, \mathsf{Q_2}, \mathsf{Q_1}, \mathsf{P_3b}, \mathsf{P_3a}, \mathsf{A}, \mathsf{Hi}, \mathsf{Hg}, \mathsf{P_2}, \mathsf{Q'}, \mathsf{PJ}, J^{\Delta}, \mathsf{P_1}, J$

B.10.2.2.2 Proof of $\mathbf{FT_2ii}$

▷ Prove subresult $\mathsf{Q_3}$ **:** $\mathsf{G} = K^{\Delta}$
= (FT$_1$iii G).equal_subs_tran (GC$_2$eq Q)

▷ Prove $\mathsf{FT_2ii}$ **:** K^{Δ}.is_galois_group
= GAL$_1$ Q$_3$ G.**2**

B.10.2.2.3 Proof of $\mathbf{FT_2iii}$

▷ Prove $\mathsf{FT_2iii}$ **:** $K = K^{\Delta\nabla}$
= Q.equal_subs_symm.equal_subs_tran (GC$_1$eq Q$_3$)

▷ Discharge $\mathsf{Q_3}, \mathsf{Q}, \mathsf{G}$

▷ Discharge to prove as claimed $\mathsf{FT_2}$
: $\{\forall K : \text{galois\_subfield } L\}$ $[\delta\mathsf{ft_2i} = \{\forall J : K\text{-subfield}\}$ J.is_galois_subfield $L]$
$[\delta\mathsf{ft_2ii} = K^{\Delta}.\text{is\_galois\_group}]$ $[\delta\mathsf{ft_2iii} = K = K^{\Delta\nabla}]$ $\bigwedge_3$ ft$_2$i ft$_2$ii ft$_2$iii
using
pair$_3$' FT$_2$i FT$_2$ii FT$_2$iii
: $\bigwedge_3$ ($\{\forall J : K\text{-subfield}\}$ J.is_galois_subfield L) K^{Δ}.is_galois_group $(K = K^{\Delta\nabla})$

▷ Discharge K

B.10.2.3 Lemma 2

▷ Introduce K **:** galois_subfield L; let $\mathsf{G} = K^{\Delta}$ **:** subgroup_of (Aut L.**1**) and
introduce J **:** K-subfield

B.10.2.3.1 Statement of Lemma 2

▷ Let $\mathsf{p}_1 = J^{\Delta}.\mathsf{normal\_in\ G} : \mathsf{prop}$

▷ Let $\mathsf{p}_2 = (\mathsf{G}\restriction J) \subseteq (\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps} : \mathsf{prop}$

▷ We want to prove $\mathsf{LEMMA_2i}$

$: (J^{\Delta}.\mathsf{normal\_in\ G}) \to (\mathsf{G}\restriction J) = (\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps}$

▷ We want to prove $\mathsf{LEMMA_2ii} : ((\mathsf{G}\restriction J) \cong (\mathsf{aut}\ J\ K)) \to J^{\Delta}.\mathsf{normal\_in\ G}$

B.10.2.3.2 Preliminaries

▷ Prove subresult $\mathsf{P_0\_G} : \mathsf{G.is\_galois\_group}$
$= \mathsf{FT_2ii}\ K$

▷ Prove subresult $\mathsf{P_0\_J} : J.\mathsf{is\_-subfield}\ \mathsf{P_0\_G}^{\nabla}$
$= \mathsf{SUBF}_1|K|\mathsf{G}^{\nabla}\ (\mathsf{FT_2iii}\ K)\ J$

▷ Prove subresult $\mathsf{Q_{0\_1}} : (\mathsf{G}\restriction J) = (\mathsf{G}^{\nabla}\text{-}\mathsf{fix\_hom}\ J)$
$= \mathsf{THM\_B}_1\ \mathsf{P_0\_G}\ \mathsf{P_0\_J}$

▷ Prove subresult $\mathsf{Q_{0\_2}} : (\mathsf{G}^{\nabla}\text{-}\mathsf{fix\_hom}\ J) = (K\text{-}\mathsf{fix\_hom}\ J)$
$= \mathsf{ALG}_1\ (\mathsf{FT_2iii}\ K).\mathsf{equal\_subs\_symm}\ J$

▷ Prove subresult $\mathsf{P_0} : (\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps} \subseteq (\mathsf{G}\restriction J)$
$= (\mathsf{ALG}_4\ J\ K).\mathsf{subs\_tran}\ (\mathsf{Q_{0\_1}.equal\_subs\_tran}\ \mathsf{Q_{0\_2}}).\mathsf{snd}$

▷ Prove subresult $\mathsf{Q} : J = J^{\Delta\nabla}$
$= \mathsf{FT_2iii}\ (\mathsf{FT_2i}\ K\ J)$

▷ Let $\mathsf{p} = \{\forall g : \mathsf{G}\}\ \{\forall x : J\}\ (g\ x) \in J^{\Delta\nabla} : \mathsf{prop}$

B.10.2.3.3 The first two implications, $\mathsf{p}_1 \leftrightarrow \mathsf{p}$

▷ We want to prove $\mathsf{LEMMA_2iii}$

$: (J^{\Delta}.\mathsf{normal\_in\ G}) \to \{\forall g : \mathsf{G}\}\ \{\forall x : J\}\ (g\ x) \in J^{\Delta\nabla}$

▷ Suppose $H : J^{\Delta}.\mathsf{normal\_in\ G}$; introduce $g : \mathsf{G}$; $x : J$ and $h : J^{\Delta}$

▷ Prove subresult $\mathsf{Q_1} : (h\ (g\ x))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (h \circ g\ x)$
$= (\mathsf{rewrite\_o}\ h\ g\ x).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q_{2\_1}} : (h \circ g)\ (=|((\mathsf{perm}\ F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}}))_{\cdot\mathbf{1}})\ (\mathsf{id} \circ (h \circ g))$
$= (\mathsf{id\_o}\ (h \circ g)).\mathsf{symm}$

▷ Prove subresult $\mathsf{Q_{2\_2}} : (\mathsf{id} \circ (h \circ g)) = ((g \circ g^{-1}) \circ (h \circ g))$
$= g.\mathsf{o\_inv.symm.o}_1\ (h \circ g)$

▷ Prove subresult $\mathsf{Q}_{2\text{-}3} : ((g \circ g^{-1}) \circ (h \circ g)) = (g \circ (g^{-1} \circ (h \circ g)))$
$= \mathsf{o\_assoc}\ g\ g^{-1}\ (h \circ g)$

▷ Prove subresult $\mathsf{Q}_2 : (h \circ g\ x) = (g \circ (g^{-1} \circ (h \circ g))\ x)$
$= \mathsf{Q}_{2\text{-}1}.\mathsf{tran}\ (\mathsf{Q}_{2\text{-}2}.\mathsf{tran}\ \mathsf{Q}_{2\text{-}3})\ x$

▷ Prove subresult $\mathsf{Q}_3 : (g \circ (g^{-1} \circ (h \circ g))\ x) = (g\ (g^{-1} \circ (h \circ g)\ x))$
$= \mathsf{rewrite\_o}\ g\ (g^{-1} \circ (h \circ g))\ x$

▷ Prove subresult $\mathsf{Q}_4\text{'} : (g^{-1} \circ (h \circ g)\ x) = x$
$= (H\ g\ h).\mathsf{fst}\ x$

▷ Prove subresult $\mathsf{Q}_4 : ((g^{-1} \circ (h \circ g)).\mathsf{fixes}\ x) \to (g\ (g^{-1} \circ (h \circ g)\ x)) = (g\ x)$
$= [\lambda H' : (g^{-1} \circ (h \circ g)).\mathsf{fixes}\ x]\ g.\mathsf{resp}\ H'$

▷ Prove subresult $\mathsf{P}_{1\text{-}1} : (h\ (g\ x))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (g\ x)$
$= (\mathsf{Q}_1.\mathsf{tran}\ \mathsf{Q}_2).\mathsf{tran}\ (\mathsf{Q}_3.\mathsf{tran}\ (\mathsf{Q}_4\ \mathsf{Q}_4\text{'}))$

▷ Discharge h

▷ Prove subresult $\mathsf{P}_{1\text{-}2\text{-}1} : g \in (\mathsf{Aut}\ L)$
$= \mathsf{G}_{\cdot\mathbf{2}}\ g$

▷ Prove subresult $\mathsf{P}_{1\text{-}2\text{-}2} : x \in L$
$= J.\mathsf{sub\_subs}\ x$

▷ Prove subresult $\mathsf{P}_{1\text{-}2\text{-}3} : (\mathsf{make}\ \mathsf{P}_{1\text{-}2\text{-}1}\ x) \in L$
$= \mathsf{AUTc}\ (\mathsf{make}\ \mathsf{P}_{1\text{-}2\text{-}1})\ (\mathsf{make}\ \mathsf{P}_{1\text{-}2\text{-}2})$

▷ Prove subresult $\mathsf{P}_{1\text{-}2\text{-}4} : (\mathsf{make}\ \mathsf{P}_{1\text{-}2\text{-}1}\ x) = (g\ x)$
$= \mathsf{refl}\ (g_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}}\ x)$

▷ Prove subresult $\mathsf{P}_{1\text{-}2} : (g\ x) \in L$
$= \mathsf{eq\_closed}\ \mathsf{P}_{1\text{-}2\text{-}4}\ \mathsf{P}_{1\text{-}2\text{-}3}$

▷ Prove subresult $\mathsf{P}_1 : (g\ x) \in J^{\Delta\nabla}$
$= \mathsf{pair}\ \mathsf{P}_{1\text{-}1}\ \mathsf{P}_{1\text{-}2}$

▷ Discharge x, g, H

▷ Suppose $H : \mathsf{p}$; introduce $g : \mathsf{G}$; $h : J^{\Delta}$ and $x : J$

▷ Prove subresult $\mathsf{Q}_5 : (g^{-1} \circ (h \circ g)\ x) = (g^{-1}\ (h \circ g\ x))$
$= \mathsf{rewrite\_o}\ g^{-1}\ (h \circ g)\ x$

▷ Prove subresult $\mathsf{Q}_{6\text{-}1} : (h \circ g\ x) = (h\ (g\ x))$
$= \mathsf{rewrite\_o}\ h\ g\ x$

▷ Prove subresult $\mathsf{Q}_6\text{'} : h.\mathsf{fixes}\ (g\ x)$
$= (H\ g\ x).\mathsf{fst}\ h$

▷ Prove subresult Q_6 : $(h.\mathsf{fixes}\ (g\ x)) \to (g^{-1}\ (h \circ g\ x)) = (g^{-1}\ (g\ x))$
= $[\lambda H' : h.\mathsf{fixes}\ (g\ x)]\ g^{-1}.\mathsf{resp}\ (\mathsf{Q}_{6\text{-}1}.\mathsf{tran}\ H')$

▷ Prove subresult Q_7 : $(g^{-1}\ (g\ x))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (g^{-1} \circ g\ x)$
= $(\mathsf{rewrite\_o}\ g^{-1}\ g\ x).\mathsf{symm}$

▷ Prove subresult Q_8 : $(g^{-1} \circ g\ x) = x$
= $\mathsf{inv\_o}\ g\ x$

▷ Prove subresult $\mathsf{P}_{2\text{-}1}$: $(g^{-1} \circ (h \circ g)\ x)\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ x$
= $(\mathsf{Q}_5.\mathsf{tran}\ (\mathsf{Q}_6\ \mathsf{Q}_6'')).\mathsf{tran}\ (\mathsf{Q}_7.\mathsf{tran}\ \mathsf{Q}_8)$

▷ Discharge x

▷ Prove subresult P_2_g : $\mathsf{equal\_in}\ (\mathsf{perm}\ F)\ (\mathsf{make}\ g.\mathsf{ev.snd})\ g$
= $\mathsf{refl}\ g_{\cdot\mathbf{1}}$

▷ Prove subresult P_2_h : $\mathsf{equal\_in}\ (\mathsf{perm}\ F)\ (\mathsf{make}\ h.\mathsf{ev.snd})\ h$
= $\mathsf{refl}\ h_{\cdot\mathbf{1}}$

▷ Prove subresult $\mathsf{P}_{2\text{-}2\text{-}1}$: $g^{-1} \in (\mathsf{Aut}\ L)$
= $\mathsf{eq\_closed}\ (\mathsf{inv\_resp}\ \mathsf{P}_2\_\mathsf{g})\ g.\mathsf{ev.snd.inv\_closed}$

▷ Prove subresult $\mathsf{P}_{2\text{-}2\text{-}2}$: $(h \circ g) \in (\mathsf{Aut}\ L)$
= $\mathsf{eq\_closed}\ (\mathsf{o\_resp}\ \mathsf{P}_2\_\mathsf{h}\ \mathsf{P}_2\_\mathsf{g})\ (h.\mathsf{ev.snd.o\_closed}\ g.\mathsf{ev.snd})$

▷ Prove subresult $\mathsf{P}_{2\text{-}2}$: $(g^{-1} \circ (h \circ g)) \in (\mathsf{Aut}\ L)$
= $\mathsf{P}_{2\text{-}2\text{-}1}.\mathsf{o\_closed}\ \mathsf{P}_{2\text{-}2\text{-}2}$

▷ Prove subresult P_2 : $(g^{-1} \circ (h \circ g)) \in J^{\Delta}$
= $\mathsf{pair}\ \mathsf{P}_{2\text{-}1}\ \mathsf{P}_{2\text{-}2}$

▷ Discharge h, g, H

▷ Prove as claimed $\mathsf{LEMMA}_2\mathsf{iii}$
: $(J^{\Delta}.\mathsf{normal\_in}\ \mathsf{G}) \to \{\forall g : \mathsf{G}\}\ \{\forall x : J\}\ (g\ x) \in J^{\Delta\nabla}$
= P_1

B.10.2.3.4 The third implication, $\mathsf{p}_2 \to \mathsf{p}$

▷ Suppose H : p_2

▷ Introduce g : G and x : J

▷ Prove subresult $\mathsf{P}_{3\text{-}1}$: $(\mathsf{restriction\_perm}\ J\ g) \in (\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps}$
= $(\mathsf{AA}_4\ \mathsf{G}\ (\mathsf{restriction\_perm}\ J)\ (\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps}).\mathsf{fst}\ H\ g$

▷ Prove subresult $\mathsf{P}_{3\text{-}2}$: $(\mathsf{make}\ \mathsf{P}_{3\text{-}1\cdot\mathbf{1}}.\mathsf{ev.snd}\ x) \in J$
= $\mathsf{AUTc}\ (\mathsf{make}\ \mathsf{P}_{3\text{-}1\cdot\mathbf{1}}.\mathsf{ev.snd})\ x$

▷ Prove subresult Q_{10} : (restriction_perm J g x) = ($\mathsf{P}_{3\text{-}1\cdot\mathbf{1}\cdot\mathbf{1}}$ x)
= $\mathsf{P}_{3\text{-}1\cdot\mathbf{2}}$ x

▷ Prove subresult Q_{11} : (restriction_perm J g x) = (make $\mathsf{P}_{3\text{-}1\cdot\mathbf{1}}$.ev.snd x)
= Q_{10}

▷ Prove subresult Q_{12} : (g x) = (restriction_perm J g x)
= REST$_1$ J g.rep x

▷ Prove subresult P_3' : (g x) $\in$ J
= eq_closed (Q_{12}.tran Q_{11}).symm $\mathsf{P}_{3\text{-}2}$

▷ Prove subresult P_3 : (g x) $\in$ $J^{\triangle\triangledown}$
= Q.fst P_3'

▷ Discharge x, g, H

B.10.2.3.5 The fourth implication, $\mathsf{p} \to \mathsf{p}_2$

▷ Suppose H : p

▷ Introduce g_1, g_2 : perm F; suppose P_g_2 : $g_2 \in \mathsf{G}$ and Qg : $(g_1 \circ g_2) = \mathsf{id}$

▷ We want to prove subresult little_lemma : $((g_1{\restriction}J) \circ (g_2{\restriction}J)) = \mathsf{id}$

▷ Introduce x : F

▷ Suppose Hn : $x \notin J$

▷ Prove subresult Q_{13} : $((g_1{\restriction}J) \circ (g_2{\restriction}J)\ x) = (g_1{\restriction}J\ (g_2{\restriction}J\ x))$
= rewrite_compose $(g_1{\restriction}J)$ $(g_2{\restriction}J)$ x

▷ Prove subresult Q_{17} : $(g_1{\restriction}J\ (g_2{\restriction}J\ x)) = (g_1{\restriction}J\ x)$
= $(g_1{\restriction}J)$.resp (REST$_2$ J g_2 Hn)

▷ Prove subresult Q_{18} : $(g_1{\restriction}J\ x) = x$
= REST$_2$ J g_1 Hn

▷ Prove subresult Cn : $((g_1{\restriction}J) \circ (g_2{\restriction}J)\ x)\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ x$
= Q_{13}.tran (Q_{17}.tran Q_{18})

▷ Suppose Hy : $x \in J$

▷ Prove subresult Q_{14} : $(g_1{\restriction}J\ (g_2{\restriction}J\ Hy)) = (g_1{\restriction}J\ (g_2\ Hy))$
= $(g_1{\restriction}J)$.resp (REST$_1$ J g_2 Hy).symm

▷ Prove subresult Q_{15}
: $(g_1{\restriction}J\ (\mathsf{Q}.\mathsf{snd}\ (H\ P\_g_2\ Hy)))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}})\ (g_1\ (\mathsf{Q}.\mathsf{snd}\ (H\ P\_g_2\ Hy)))$
= (REST$_1$ J g_1 (Q.snd $(H\ P\_g_2\ Hy)$)).symm

▷ Prove subresult Q_{16} : $(g_1\ (g_2\ x))\ (=|F)\ (g_1 \circ g_2\ x)$
= $(\mathsf{rewrite\_o}\ g_1\ g_2\ x).\mathsf{symm}$

▷ Prove subresult Cy : $((g_1{\restriction}J) \circ (g_2{\restriction}J)\ x)\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ ((\mathsf{id})_{\cdot\mathbf{1}\cdot\mathbf{1}}\ x)$
= $(\mathsf{Q}_{13}.\mathsf{tran}\ (\mathsf{Q}_{14}.\mathsf{tran}\ \mathsf{Q}_{15})).\mathsf{tran}\ (\mathsf{Q}_{16}.\mathsf{tran}\ (Qg\ x))$

▷ Discharge Hy, Hn

▷ Discharge to prove subresult as claimed $\mathsf{little\_lemma}$: $((g_1{\restriction}J) \circ (g_2{\restriction}J)) = \mathsf{id}$
using
$\mathsf{case}\ (x \in_?\ J)\ \mathsf{Cy}\ \mathsf{Cn} : ((g_1{\restriction}J) \circ (g_2{\restriction}J)).\mathsf{fixes}\ x$

▷ Discharge x

▷ Discharge Qg, P_g_2, g_2, g_1

▷ Introduce g : $\mathsf{G}{\restriction}J$ and let $g' = (g.\mathsf{ev})_{\cdot\mathbf{1}}$: G

▷ Prove subresult $\mathsf{P\_iso}_1$: $((g'.\mathsf{rep}^{-1}{\restriction}J) \circ (g'.\mathsf{rep}{\restriction}J)) = \mathsf{id}$
= $\mathsf{little\_lemma}\ g'.\mathsf{rep}^{-1}\ g'.\mathsf{rep}\ g'.\mathsf{ev}\ (\mathsf{inv\_o}\ g'.\mathsf{rep})$

▷ Prove subresult $\mathsf{P\_iso}_2$: $((g'.\mathsf{rep}{\restriction}J) \circ (g'.\mathsf{rep}^{-1}{\restriction}J)) = \mathsf{id}$
= $\mathsf{little\_lemma}\ g'.\mathsf{rep}\ g'.\mathsf{rep}^{-1}\ (\mathsf{inv\_closed}|(\mathsf{perm}\ F)|\mathsf{G}\ g')\ g'.\mathsf{rep.o\_inv}$

▷ Prove subresult $\mathsf{P\_iso}$: $(g'.\mathsf{rep}{\restriction}J) \in (\mathsf{iso}\ F\ F)$
= $(g'.\mathsf{rep}^{-1}{\restriction}J, \mathsf{pair}\ \mathsf{P\_iso}_1\ \mathsf{P\_iso}_2 : (g'.\mathsf{rep}{\restriction}J) \in (\mathsf{iso}\ F\ F))$

▷ Prove subresult $\mathsf{P\_fix}$: $(\mathsf{make}\ \mathsf{P\_iso}) \in (\mathsf{fixing\_group}\ K)$
= $[\lambda x : K]\ (\mathsf{REST}_1\ J\ g'.\mathsf{rep}\ (\mathsf{subs\_sub}\ J\ x)).\mathsf{symm.tran}\ (g'.\mathsf{ev.fst}\ x)$

▷ Prove subresult $\mathsf{P\_Perm}$: $(\mathsf{make}\ \mathsf{P\_iso}) \in (\mathsf{Perm}\ J)$
= $\mathsf{REST}_5\ J\ g'.\mathsf{rep}$

▷ Introduce x, y : J

▷ Prove subresult Q_{19}
: $(g'.\mathsf{rep}{\restriction}J\ (x.\mathsf{plus\_closed}\ y))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}})\ (g'.\mathsf{rep}\ (x.\mathsf{plus\_closed}\ y))$
= $(\mathsf{REST}_1\ J\ g'.\mathsf{rep}\ (x.\mathsf{plus\_closed}\ y)).\mathsf{symm}$

▷ Prove subresult Q_{20} : $(g'.\mathsf{rep}\ (x + y)) = ((g'.\mathsf{rep}\ x) + (g'.\mathsf{rep}\ y))$
= $g'.\mathsf{ev.snd.snd.fst}\ (J.\mathsf{sub\_subs}\ x)\ (J.\mathsf{sub\_subs}\ y)$

▷ Prove subresult $\mathsf{Q}_{21}\_\mathsf{x}$: $x \in (g'.\mathsf{rep.agree}\ (g'.\mathsf{rep}{\restriction}J))$
= $\mathsf{REST}_1\ J\ g'.\mathsf{rep}\ x$

▷ Prove subresult $\mathsf{Q}_{21}\_\mathsf{y}$: $y \in (g'.\mathsf{rep.agree}\ (g'.\mathsf{rep}{\restriction}J))$
= $\mathsf{REST}_1\ J\ g'.\mathsf{rep}\ y$

▷ Prove subresult Q_{21} : $((g'.\mathsf{rep}\ x) + (g'.\mathsf{rep}\ y)) = ((g'.\mathsf{rep}{\restriction}J\ x) + (g'.\mathsf{rep}{\restriction}J\ y))$
= $\mathsf{plus\_resp}\ \mathsf{Q}_{21}\_\mathsf{x}\ \mathsf{Q}_{21}\_\mathsf{y}$

▷ Prove subresult $\mathsf{P\_plus}$
: $(g'.\mathsf{rep}{\restriction}J\ (x.\mathsf{plus\_closed}\ y))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}})\ ((g'.\mathsf{rep}{\restriction}J\ x) + (g'.\mathsf{rep}{\restriction}J\ y))$
$= \mathsf{Q}_{19}.\mathsf{tran}\ (\mathsf{Q}_{20}.\mathsf{tran}\ \mathsf{Q}_{21})$

▷ Prove subresult Q_{22}
: $(g'.\mathsf{rep}{\restriction}J\ (x.\mathsf{times\_closed}\ y))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}})\ (g'.\mathsf{rep}\ (x.\mathsf{times\_closed}\ y))$
$= (\mathsf{REST}_1\ J\ g'.\mathsf{rep}\ (x.\mathsf{times\_closed}\ y)).\mathsf{symm}$

▷ Prove subresult Q_{23} : $(g'.\mathsf{rep}\ (x \times y)) = ((g'.\mathsf{rep}\ x) \times (g'.\mathsf{rep}\ y))$
$= g'.\mathsf{ev}.\mathsf{snd}.\mathsf{snd}.\mathsf{snd}\ (J.\mathsf{sub\_subs}\ x)\ (J.\mathsf{sub\_subs}\ y)$

▷ Prove subresult Q_{24} : $((g'.\mathsf{rep}\ x) \times (g'.\mathsf{rep}\ y)) = ((g'.\mathsf{rep}{\restriction}J\ x) \times (g'.\mathsf{rep}{\restriction}J\ y))$
$= \mathsf{times\_resp}\ \mathsf{Q}_{21}\_x\ \mathsf{Q}_{21}\_y$

▷ Prove subresult $\mathsf{P\_times}$
: $(g'.\mathsf{rep}{\restriction}J\ (x.\mathsf{times\_closed}\ y))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}})\ ((g'.\mathsf{rep}{\restriction}J\ x) \times (g'.\mathsf{rep}{\restriction}J\ y))$
$= \mathsf{Q}_{22}.\mathsf{tran}\ (\mathsf{Q}_{23}.\mathsf{tran}\ \mathsf{Q}_{24})$

▷ Discharge y, x

▷ Prove subresult $\mathsf{P\_aut}$: $(\mathsf{make\ P\_iso}) \in (\mathsf{aut}\ J\ K)$
$= \mathsf{pair\ P\_fix\ (pair\ P\_Perm\ (pair\ P\_plus\ P\_times))}$

▷ Prove subresult $\mathsf{P\_re}$
: $(\mathsf{restriction\_perm}\ J\ (g.\mathsf{ev})_{\cdot\mathbf{1}}.\mathsf{rep}) \in (\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps}$
$= (\mathsf{make\ P\_aut}, \mathsf{refl}\ (\mathsf{restriction\_perm}\ J\ (g.\mathsf{ev})_{\cdot\mathbf{1}}.\mathsf{rep}) :$
$(\mathsf{restriction\_perm}\ J\ (g.\mathsf{ev})_{\cdot\mathbf{1}}.\mathsf{rep}) \in (\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps})$
▷ Prove subresult P_4 : $g \in (\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps}$
$= \mathsf{eq\_closed}\ (g.\mathsf{ev})_{\cdot\mathbf{2}}.\mathsf{symm}\ \mathsf{P\_re}$

▷ Discharge g, H

B.10.2.3.6 The end of the proof

▷ Prove as claimed $\mathsf{LEMMA}_2\mathsf{i}$
: $(J^{\Delta}.\mathsf{normal\_in}\ \mathsf{G}) \to (\mathsf{G}{\restriction}J) = (\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps}$
$= [\lambda H : \mathsf{p}_1]\ \mathsf{pair}\ (\mathsf{P}_4\ (\mathsf{P}_1\ H))\ \mathsf{P}_0$

▷ Suppose H : $(\mathsf{G}{\restriction}J) \cong (\mathsf{aut}\ J\ K)$

▷ Prove subresult $\mathsf{P}_{5\text{-}1}$: $(\mathsf{G}{\restriction}J).\mathsf{is\_finite}$
$= \mathsf{THM\_B}_0\ \mathsf{P}_0\_\mathsf{G}\ \mathsf{P}_0\_\mathsf{J}$

▷ Prove subresult $\mathsf{P}_{6\text{-}1}$: $\{\forall x_1, x_2 \mid \mathsf{aut}\ J\ K\}$
$((\mathsf{rep\_map}\ (\mathsf{iso}\ F\ F)\ x_1) = (\mathsf{rep\_map}\ (\mathsf{iso}\ F\ F)\ x_2)) \to x_1 = x_2$
$= [\lambda x_1, x_2 \mid \mathsf{aut}\ J\ K]\ [\lambda Q : (\mathsf{rep\_map}\ (\mathsf{iso}\ F\ F)\ x_1) = (\mathsf{rep\_map}\ (\mathsf{iso}\ F\ F)\ x_2)]\ Q$

▷ Prove subresult $\mathsf{P}_{6\text{-}2}$: (aut J K).as_a_subset_of_maps $\cong$ (aut J K)
= AA_5 (aut J K) (rep_map (iso F F)) $\mathsf{P}_{6\text{-}1}$

▷ Prove subresult P_6 : (aut J K).as_a_subset_of_maps $\cong$ (G$\restriction J$)
= $\mathsf{P}_{6\text{-}2}$.eqsize_tran H.eqsize_symm

▷ Prove subresult P_5 : (aut J K).as_a_subset_of_maps.is_finite
= EQSIZE_0 P_6.eqsize_symm $\mathsf{P}_{5\text{-}1}$

▷ Prove subresult H' : (aut J K).as_a_subset_of_maps = (G$\restriction J$)
= FIN_3 P_0 P_5 P_6

▷ Discharge to prove as claimed LEMMA_2ii
: ((G$\restriction J$) $\cong$ (aut J K)) $\to$ J^{Δ}.normal_in G
using
P_2 (P_3 H'.snd) : p_1

▷ Discharge H

▷ Discharge H', P_5, P_6, $\mathsf{P}_{6\text{-}2}$, $\mathsf{P}_{6\text{-}1}$, $\mathsf{P}_{5\text{-}1}$, P_4, P_re, P_aut, P_times, Q_{24}, Q_{23}, Q_{22}, P_plus, Q_{21}, Q_{21}_y, Q_{21}_x, Q_{20}, Q_{19}, P_Perm, P_fix, P_iso, $\mathsf{P}_{\text{iso}\text{-}2}$, $\mathsf{P}_{\text{iso}\text{-}1}$, g', little_lemma, Cy, Q_{16}, Q_{15}, Q_{14}, Cn, Q_{18}, Q_{17}, Q_{13}, P_3, P_3', Q_{12}, Q_{11}, Q_{10}, $\mathsf{P}_{3\text{-}2}$, $\mathsf{P}_{3\text{-}1}$, P_2, $\mathsf{P}_{2\text{-}2}$, $\mathsf{P}_{2\text{-}2\text{-}2}$, $\mathsf{P}_{2\text{-}2\text{-}1}$, P_2_h, P_2_g, $\mathsf{P}_{2\text{-}1}$, Q_8, Q_7, Q_6, Q_6', $\mathsf{Q}_{6\text{-}1}$, Q_5, P_1, $\mathsf{P}_{1\text{-}2}$, $\mathsf{P}_{1\text{-}2\text{-}4}$, $\mathsf{P}_{1\text{-}2\text{-}3}$, $\mathsf{P}_{1\text{-}2\text{-}2}$, $\mathsf{P}_{1\text{-}2\text{-}1}$, $\mathsf{P}_{1\text{-}1}$, Q_4, Q_4', Q_3, Q_2, $\mathsf{Q}_{2\text{-}3}$, $\mathsf{Q}_{2\text{-}2}$, $\mathsf{Q}_{2\text{-}1}$, Q_1, p, Q, P_0, $\mathsf{Q}_{0\text{-}2}$, $\mathsf{Q}_{0\text{-}1}$, P_0_J, P_0_G, p_2, p_1, J, G, K

▷ Discharge but keep L

B.10.2.4 Proof of final part of the fundamental theorem

▷ Introduce K : galois_subfield L and J : K-subfield

▷ Let G = FT_2ii K : galois_group L

▷ Prove subresult $\mathsf{Q}_{1\text{-}1}$: $K = K^{\Delta\nabla}$
= FT_2iii K

▷ Let P_0 = $\mathsf{SUBF}_1|L|K|K^{\Delta\nabla}$ $\mathsf{Q}_{1\text{-}1}$ J : J.is_-subfield G$^{\nabla}$

B.10.2.4.1 Proof of $\mathbf{FT_3}$i

▷ Suppose H : K.is_galois_subfield J

▷ Let $\mathsf{G'} = \mathsf{FT}_2\mathsf{ii}\ H$: galois_group J

▷ Prove subresult $\mathsf{Q}_{1\text{-}2}$: $K = H$
= equal_subs_refl H

▷ Prove subresult $\mathsf{Q}_{1\text{-}3}$: $H = H.(\mathsf{aut}\ J).(\mathsf{fix}\ J)$
= $\mathsf{FT}_2\mathsf{iii}\ H$

▷ Prove subresult $\mathsf{Q}_{1\text{-}4}$: $\mathsf{G}.(\mathsf{fix}\ L) = \mathsf{G'}.(\mathsf{fix}\ J)$
= $\mathsf{Q}_{1\text{-}1}$.equal_subs_symm.equal_subs_tran ($\mathsf{Q}_{1\text{-}2}$.equal_subs_tran $\mathsf{Q}_{1\text{-}3}$)

▷ Prove (but keep unfrozen) subresult P_1' : P_0.is_fin_dim_over G^∇
= sub_fin_dim_over G^∇ P_0

▷ Prove subresult P_1
: $[\delta K = \mathsf{G'}^\nabla]\ \langle \exists n : \mathbb{N} \rangle\ \langle \exists \boldsymbol{w} : F^\wedge n \rangle\ (J_{.\mathbf{1}\cdot\mathbf{1}}.\mathsf{has\_basis\_over}\ K\ \boldsymbol{w}) \wedge (\mathsf{G'} \cong \boldsymbol{w})$
= $\mathsf{THM\text{-}A}_3$ G'

▷ Prove subresult P_2 : $J.\mathsf{is\_fin\_dim\_over}\ \mathsf{G'}.(\mathsf{fix}\ J)$
= $\mathsf{FINDIM}_2\ J\ \mathsf{Q}_{1\text{-}4}\ \mathsf{P}_1$'

▷ Let $\mathsf{n} = \mathsf{P}_{1\cdot\mathbf{1}}$: $\mathbb{N}$

▷ Let $\boldsymbol{w} = \mathsf{P}_{1\cdot\mathbf{2}\cdot\mathbf{1}}$: $F^\wedge\mathsf{n}$

▷ Let $\mathsf{m} = \dim\ \mathsf{P}_1$' : $\mathbb{N}$

▷ Let $\boldsymbol{v} = \mathsf{basis}\ \mathsf{P}_1$' : $J^\wedge\mathsf{m}$

▷ Prove subresult Q_2 : $(K^\Delta \restriction J) \cong \boldsymbol{v}$
= $\mathsf{THM\text{-}B}_2$ G P_0

▷ Prove subresult Q_1 : $\boldsymbol{v} \cong \boldsymbol{w}$
= $\mathsf{FINDIM}_4\mathsf{b}\ \boldsymbol{v}\ \boldsymbol{w}\ (\mathsf{FINDIM}_2\mathsf{b}\ J\ \mathsf{Q}_{1\text{-}4}\ \mathsf{P}_1\text{'}_{\cdot\mathbf{2}\cdot\mathbf{2}})\ \mathsf{P}_{1\cdot\mathbf{2}\cdot\mathbf{2}}.\mathsf{fst}$

▷ Prove $\mathsf{FT}_3\mathsf{i}$: $J^\Delta.\mathsf{normal\_in}\ K^\Delta$
= $\mathsf{LEMMA}_2\mathsf{ii}\ K\ J\ ((\mathsf{Q}_2.\mathsf{eqsize\_tran}\ \mathsf{Q}_1).\mathsf{eqsize\_tran}\ \mathsf{P}_{1\cdot\mathbf{2}\cdot\mathbf{2}}.\mathsf{snd.eqsize\_symm})$

▷ Discharge $\mathsf{Q}_1, \mathsf{Q}_2, \boldsymbol{v}, \mathsf{m}, \boldsymbol{w}, \mathsf{n}, \mathsf{P}_2, \mathsf{P}_1, \mathsf{P}_1\text{'}, \mathsf{Q}_{1\text{-}4}, \mathsf{Q}_{1\text{-}3}, \mathsf{Q}_{1\text{-}2}, \mathsf{G'}, H$

B.10.2.4.2 Proof of $\mathbf{FT}_3$ii

▷ Suppose H : $J^\Delta.\mathsf{normal\_in}\ K^\Delta$

▷ We want to prove $\mathsf{FT}_3\mathsf{ii}$: $K.\mathsf{is\_galois\_subfield}\ J$

▷ Prove subresult P_1 : $(K^\Delta \restriction J) = (\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps}$
= $\mathsf{LEMMA}_2\mathsf{i}\ K\ J\ H$

▷ We want to prove subresult $\mathsf{P}_{3\text{-}2}$: $(\mathsf{fix}\ J\ (\mathsf{aut}\ J\ K)) \subseteq K^{\Delta\nabla}$

▷ Prove subresult $\mathsf{P}_{2\text{-}1\text{-}1}$: $(\mathsf{G}\restriction J).\mathsf{is\_finite}$
$= \mathsf{THM\_B}_0\ \mathsf{G}\ \mathsf{P}_0$

▷ Prove subresult $\mathsf{P}_{2\text{-}1\text{-}2}$: $(\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps.is\_finite}$
$= \mathsf{FIN}_4\ \mathsf{P}_1\ \mathsf{P}_{2\text{-}1\text{-}1}$

▷ Prove subresult $\mathsf{P}_{2\text{-}1\text{-}3}$: $\{\forall x_1, x_2 \mid \mathsf{aut}\ J\ K\}$
$((\mathsf{rep\_map}\ (\mathsf{iso}\ F\ F)\ x_1) = (\mathsf{rep\_map}\ (\mathsf{iso}\ F\ F)\ x_2)) \to x_1 = x_2$
$= [\lambda x_1, x_2 \mid \mathsf{aut}\ J\ K]\ [\lambda Q : (\mathsf{rep\_map}\ (\mathsf{iso}\ F\ F)\ x_1) = (\mathsf{rep\_map}\ (\mathsf{iso}\ F\ F)\ x_2)]\ Q$

▷ Prove subresult $\mathsf{P}_{2\text{-}1\text{-}4}$: $(\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps} \cong (\mathsf{aut}\ J\ K)$
$= \mathsf{AA}_5\ (\mathsf{aut}\ J\ K)\ (\mathsf{rep\_map}\ (\mathsf{iso}\ F\ F))\ \mathsf{P}_{2\text{-}1\text{-}3}$

▷ Prove subresult $\mathsf{P}_{3\text{-}1}$: $K \subseteq (\mathsf{fix}\ J\ (\mathsf{aut}\ J\ K))$
$= \mathsf{GC}_4\ (\mathsf{subs\_sub}\ J)$

▷ Introduce x : $\mathsf{fix}\ J\ (\mathsf{aut}\ J\ K)$ and g : K^Δ

▷ Prove subresult $\mathsf{P}_{3\text{-}2\text{-}2\text{-}1}$: $x \in J$
$= x.\mathsf{ev.snd}$

▷ Prove subresult $\mathsf{P}_{3\text{-}2\text{-}2}$: $x \in L$
$= J.\mathsf{sub\_subs}\ \mathsf{P}_{3\text{-}2\text{-}2\text{-}1}$

▷ Let $\mathsf{gbarJ} = g\restriction J$: $\mathsf{map}\ F\ F$

▷ Prove (but keep unfrozen) subresult $\mathsf{P}_{3\text{-}2\text{-}3}$: $\mathsf{gbarJ} \in (K^\Delta\restriction J)$
$= (g, \mathsf{refl\ gbarJ} : \mathsf{gbarJ} \in (K^\Delta\restriction J))$

▷ Prove subresult $\mathsf{P}_{3\text{-}2\text{-}4}$: $\mathsf{gbarJ} \in (\mathsf{aut}\ J\ K).\mathsf{as\_a\_subset\_of\_maps}$
$= \mathsf{P}_1.\mathsf{fst}\ \mathsf{P}_{3\text{-}2\text{-}3}$

▷ Let $g' = \mathsf{P}_{3\text{-}2\text{-}4\cdot\mathbf{1}}$: $\mathsf{aut}\ J\ K$

▷ Prove subresult $\mathsf{Q}_{2\text{-}1}$: $(g\ x) = (\mathsf{gbarJ}\ x)$
$= \mathsf{REST}_1\ J\ g\ \mathsf{P}_{3\text{-}2\text{-}2\text{-}1}$

▷ Prove subresult $\mathsf{Q}_{2\text{-}2}$: $(\mathsf{gbarJ}\ x) = (g'\ x)$
$= \mathsf{P}_{3\text{-}2\text{-}4\cdot\mathbf{2}}\ x$

▷ Prove subresult $\mathsf{Q}_{2\text{-}3}$: $(g'\ x) = x$
$= x.\mathsf{ev.fst}\ g'$

▷ Prove subresult P_3' : $(\mathsf{gbarJ}\ x)\ (=\restriction F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ x$
$= \mathsf{Q}_{2\text{-}2}.\mathsf{tran}\ \mathsf{Q}_{2\text{-}3}$

▷ Prove subresult Q_2 : $g.\mathsf{fixes}\ x$
$= \mathsf{Q}_{2\text{-}1}.\mathsf{tran}\ \mathsf{P}_3$'

▷ Discharge g, x

▷ Prove subresult as claimed $\mathsf{P}_{3\text{-}2}$: $(\mathsf{fix}\ J\ (\mathsf{aut}\ J\ K)) \subseteq K^{\Delta\nabla}$
$= [\lambda x : \mathsf{fix}\ J\ (\mathsf{aut}\ J\ K)]\ \mathsf{pair}\ (\mathsf{Q}_2\ x)\ (\mathsf{P}_{3\text{-}2\text{-}2}\ x)$

▷ Prove subresult $\mathsf{P}_{3\text{-}3}$: $(\mathsf{fix}\ J\ (\mathsf{aut}\ J\ K)) \subseteq K$
$= \mathsf{P}_{3\text{-}2}.\mathsf{subs\_tran}\ (\mathsf{FT}_2\mathsf{iii}\ K).\mathsf{snd}$

▷ Prove subresult P_3 : $K = (\mathsf{fix}\ J\ (\mathsf{aut}\ J\ K))$
$= \mathsf{pair}\ \mathsf{P}_{3\text{-}1}\ \mathsf{P}_{3\text{-}3}$

▷ Prove subresult $\mathsf{P}_{2\text{-}1}$: $(\mathsf{aut}\ J\ K).\mathsf{is\_finite}$
$= \mathsf{EQSIZE}_0\ \mathsf{P}_{2\text{-}1\text{-}4}\ \mathsf{P}_{2\text{-}1\text{-}2}$

▷ Prove subresult $\mathsf{P}_{2\text{-}2}$: $J.\mathsf{is\_fin\_dim\_over}\ (\mathsf{fix}\ J\ (\mathsf{aut}\ J\ K))$
$= \mathsf{FINDIM}_2\ J\ \mathsf{P}_3\ (\mathsf{sub\_fin\_dim\_over}\ K\ J)$

▷ Prove subresult P_2 : $(\mathsf{aut}\ J\ K).\mathsf{is\_galois\_group}$
$= \mathsf{pair}\ \mathsf{P}_{2\text{-}1}\ \mathsf{P}_{2\text{-}2}$

▷ Prove as claimed $\mathsf{FT}_3\mathsf{ii}$: $K.\mathsf{is\_galois\_subfield}\ J$
$= (\mathsf{P}_2, \mathsf{P}_3.\mathsf{equal\_subs\_symm} : K.\mathsf{is\_galois\_subfield}\ J)$

▷ Discharge $\mathsf{P}_2, \mathsf{P}_{2\text{-}2}, \mathsf{P}_{2\text{-}1}, \mathsf{P}_3, \mathsf{P}_{3\text{-}3}, \mathsf{P}_{3\text{-}2}, \mathsf{Q}_2, \mathsf{P}_3', \mathsf{Q}_{2\text{-}3}, \mathsf{Q}_{2\text{-}2}, \mathsf{Q}_{2\text{-}1}, g', \mathsf{P}_{3\text{-}2\text{-}4}$,
$\mathsf{P}_{3\text{-}2\text{-}3}, \mathsf{gbarJ}, \mathsf{P}_{3\text{-}2\text{-}2}, \mathsf{P}_{3\text{-}2\text{-}2\text{-}1}, \mathsf{P}_{3\text{-}1}, \mathsf{P}_{2\text{-}1\text{-}4}, \mathsf{P}_{2\text{-}1\text{-}3}, \mathsf{P}_{2\text{-}1\text{-}2}, \mathsf{P}_{2\text{-}1\text{-}1}, \mathsf{P}_1, H$

B.10.2.4.3 Proof of $\mathbf{FT}_3$iii

▷ Let $U = K^{\Delta} \cap J^{\Delta}$: subgroup (perm F)

▷ Suppose H : $K.\mathsf{is\_galois\_subfield}\ J$

▷ Let $G{\restriction}J = \mathsf{G}{\restriction}J$: subset (map F F)

▷ Let $G/U = \mathsf{G}/U$: subset ((perm F)/U)

▷ Let $\mathsf{GqJa} = \mathsf{G}/J^{\Delta}$: subset ((perm F)/J^{Δ})

▷ Let coercion $\mathsf{apply\_GqJa} = [\lambda g : \mathsf{GqJa}]\ g.\mathsf{rep.rep}$: GqJa → map F F

▷ Prove subresult $\mathsf{P}_{1\text{-}1\text{-}1}$: $U \subseteq J^{\Delta}$
$= [\lambda g : U]\ g.\mathsf{ev.snd}$

▷ Prove subresult $\mathsf{P}_{1\text{-}1\text{-}2\text{-}1}$: $J^{\Delta} \subseteq \mathsf{G}$
$= \mathsf{GC}_2|L|K|J\ (\mathsf{subs\_sub}\ J)$

▷ Prove subresult $\mathsf{P}_{1\text{-}1\text{-}2}$: $J^{\Delta} \subseteq U$
$= [\lambda g : J^{\Delta}]\ \mathsf{pair}\ (\mathsf{P}_{1\text{-}1\text{-}2\text{-}1}\ g)\ g.\mathsf{ev}$

▷ Prove subresult $\mathsf{P}_{1\text{-}1}$: $J^{\Delta} = U$
$= \mathsf{pair}\ \mathsf{P}_{1\text{-}1\text{-}2}\ \mathsf{P}_{1\text{-}1\text{-}1}$

▷ Prove subresult $\mathsf{P}_{1\text{-}2}$: $(\approx\text{-}J^{\Delta})$.is_same_relation $(\approx\text{-}U)$
= $\mathsf{COSET}_1|(\mathsf{perm}\ F)|J^{\Delta}|U\ \mathsf{P}_{1\text{-}1}$

▷ Let π_1 = quot_iso G $\mathsf{P}_{1\text{-}2}$: iso GqJa G/U

▷ Let $\pi_2 = \theta|L$ G J : iso G/U $G\restriction J$

▷ Prove subresult $\mathsf{P}_{3\text{-}1}$: J^{Δ}.normal_in G
= $\mathsf{FT}_3\mathsf{i}\ H$

▷ Prove subresult $\mathsf{P}_{3\text{-}2}$: $G\restriction J$ = (aut J K).as_a_subset_of_maps
= $\mathsf{LEMMA}_2\mathsf{i}\ K\ J\ \mathsf{P}_{3\text{-}1}$

▷ Let π_3 = equal_iso $\mathsf{P}_{3\text{-}2}$: iso $G\restriction J$ (aut J K).as_a_subset_of_maps

▷ Prove subresult $\mathsf{P}_{4\text{-}1}$: $\{\forall x_1, x_2 \,|\, \mathsf{aut}\ J\ K\}$
((rep_map (iso F F) x_1) = (rep_map (iso F F) x_2)) $\rightarrow x_1 = x_2$
= $[\lambda x_1, x_2 \,|\, \mathsf{aut}\ J\ K]\ [\lambda Q : (\mathsf{rep\_map}\ (\mathsf{iso}\ F\ F)\ x_1) = (\mathsf{rep\_map}\ (\mathsf{iso}\ F\ F)\ x_2)]\ Q$

▷ Let π_4 = apply_across_iso (aut J K) (rep_map (iso F F)) $\mathsf{P}_{4\text{-}1}$
: iso (aut J K).as_a_subset_of_maps (aut J K)

▷ Let $\pi = ((\pi_4 \circ \pi_3) \circ \pi_2) \circ \pi_1$: iso (K^{Δ}/J^{Δ}) (aut J K)

▷ Introduce g, h : G/J^{Δ} and x : F

▷ Let g' = $(g.\mathsf{ev})_{.\mathbf{1}}$: G

▷ Let $\mathsf{g}_1 = \pi_1\ g$: G/U

▷ Let $\mathsf{g}_2 = \pi_2\ \mathsf{g}_1$: $G\restriction J$

▷ Let $\mathsf{g}_3 = \pi_3\ \mathsf{g}_2$: (aut J K).as_a_subset_of_maps

▷ Let $\mathsf{g}_4 = \pi_4\ \mathsf{g}_3$: aut J K

▷ Prove subresult $\mathsf{Q}_{2\text{-}1}$: equal_in (aut J K) (π g) $((\pi_4 \circ \pi_3) \circ \pi_2\ \mathsf{g}_1)$
= $\mathsf{rewrite\_iso\_compose}\ ((\pi_4 \circ \pi_3) \circ \pi_2)\ \pi_1\ g$

▷ Prove subresult $\mathsf{Q}_{2\text{-}2}$: equal_in (aut J K) $((\pi_4 \circ \pi_3) \circ \pi_2\ \mathsf{g}_1)$ $(\pi_4 \circ \pi_3\ \mathsf{g}_2)$
= $\mathsf{rewrite\_iso\_compose}\ (\pi_4 \circ \pi_3)\ \pi_2\ \mathsf{g}_1$

▷ Prove subresult $\mathsf{Q}_{2\text{-}3}$: equal_in (aut J K) $(\pi_4 \circ \pi_3\ \mathsf{g}_2)$ g_4
= $\mathsf{rewrite\_iso\_compose}\ \pi_4\ \pi_3\ \mathsf{g}_2$

▷ Prove subresult conversion_lemma
: $\{\forall g_1, g_2 \,|\, \mathsf{aut}\ J\ K\}$ (equal_in (aut J K) g_1 g_2) $\rightarrow$ equal_in (map F F) g_1 g_2
= $[\lambda g_1, g_2 \,|\, \mathsf{aut}\ J\ K]\ [\lambda Q : \mathsf{equal\_in}\ (\mathsf{aut}\ J\ K)\ g_1\ g_2]\ Q$

▷ Prove subresult Q_2' : equal_in (aut J K) (π g) g_4
= $\mathsf{Q}_{2\text{-}1}.\mathsf{tran}\ (\mathsf{Q}_{2\text{-}2}.\mathsf{tran}\ \mathsf{Q}_{2\text{-}3})$

▷ Prove subresult Q_2 : equal_in (map F F) (π g) g_4
= conversion_lemma Q_2'

▷ Introduce y : J

▷ Prove subresult P_1 : $(g'^{-1} \circ g_{\cdot\mathbf{1}}) \in J^{\triangle}$
= $g_{\cdot\mathbf{2}\cdot\mathbf{2}}$

▷ Prove subresult $\mathsf{Q}_{3\text{-}1}$: $(g\ y) = (g' \circ g'^{-1}\ (g\ y))$
= (o_inv g').symm (g y)

▷ Prove subresult $\mathsf{Q}_{3\text{-}2}$: $(g' \circ g'^{-1}\ (g\ y)) = (g'\ (g'^{-1}\ (g\ y)))$
= rewrite_o g' g'^{-1} (g y)

▷ Prove subresult $\mathsf{Q}_{3\text{-}3\text{-}1}$: $(g'^{-1}\ (g_{\cdot\mathbf{1}}\ y))\ (=|F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}})\ (g'^{-1} \circ g_{\cdot\mathbf{1}}\ y)$
= (rewrite_o g'^{-1} $g_{\cdot\mathbf{1}}$ y).symm

▷ Prove subresult $\mathsf{Q}_{3\text{-}3\text{-}2}$: $(g'^{-1} \circ g_{\cdot\mathbf{1}}\ y) = y$
= P_1.fst y

▷ Prove subresult $\mathsf{Q}_{3\text{-}3}$: $(g'\ (g'^{-1}\ (g_{\cdot\mathbf{1}}\ y))) = (g'\ y)$
= g'.resp ($\mathsf{Q}_{3\text{-}3\text{-}1}$.tran $\mathsf{Q}_{3\text{-}3\text{-}2}$)

▷ Prove subresult Q_3' : (g y) (=|$F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}}$) ($g'$ y)
= ($\mathsf{Q}_{3\text{-}1}$.tran $\mathsf{Q}_{3\text{-}2}$).tran $\mathsf{Q}_{3\text{-}3}$

▷ Discharge y

▷ Prove subresult Q_3 : $(g{\restriction}J) = (g'{\restriction}J)$
= (REST$_4$ J g g').fst Q_3'

▷ Prove subresult Q_4 : $(g'{\restriction}J) = \mathsf{g}_4$
= $\mathsf{g}_{3\cdot\mathbf{2}\cdot\mathbf{2}}$

▷ Prove subresult P_2 : $(g{\restriction}J)$ (=|(map $F_{\cdot\mathbf{1}\cdot\mathbf{1}}$ $F_{\cdot\mathbf{1}\cdot\mathbf{1}}$)) ($\pi$ g)
= (Q_3.tran Q_4).tran Q_2.symm

▷ Let $\mathsf{gx} = g\ x$: F and $\mathsf{hgx} = h\ \mathsf{gx}$: F

▷ Let $\mathsf{gJ} = g{\restriction}J$: map $F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}}$ $F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}}$ and $\mathsf{hJ} = h{\restriction}J$: map $F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}}$ $F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}}$

▷ Let $\mathsf{hg} = h \circ g$: map $F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}}$ $F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}}$ and $\mathsf{hgJ} = \mathsf{hg}{\restriction}J$: map $F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}}$ $F_{\cdot\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1}}$

▷ Suppose H_1 : $x \in J$

▷ Prove subresult $\mathsf{C}_1\mathsf{Q}_1$: (hgJ x) = (hg x)
= (REST$_1$ J hg H_1).symm

▷ Prove subresult $\mathsf{C}_1\mathsf{Q}_2$: (hg x) = hgx
= rewrite_compose h g x

▷ Prove subresult $\mathsf{C_1Q_{3\text{-}1}}$: $(g'\ x) = \mathsf{gx}$
= ($\mathsf{Q_3}$' H_1).symm

▷ Prove subresult $\mathsf{P_{3\text{-}3}}$: $(g'\ x) \in J^{\triangle\triangledown}$
= LEMMA$_2$iii K J $\mathsf{P_{3\text{-}1}}$ g' H_1

▷ Prove subresult $\mathsf{P_{3\text{-}4}}$: J.is_galois_subfield L
= FT$_2$i K J

▷ Prove subresult $\mathsf{P_{3\text{-}5}}$: $J^{\triangle\triangledown} \subseteq J$
= (FT$_2$iii|L $\mathsf{P_{3\text{-}4}}$).snd

▷ Prove subresult $\mathsf{P_{3\text{-}6}}$: $(g'\ x) \in J$
= $\mathsf{P_{3\text{-}5}}$ $\mathsf{P_{3\text{-}3}}$

▷ Prove subresult $\mathsf{P_3}$: $\mathsf{gx} \in J$
= eq_closed $\mathsf{C_1Q_{3\text{-}1}}$ $\mathsf{P_{3\text{-}6}}$

▷ Prove subresult $\mathsf{C_1Q_3}$: $\mathsf{hgx} = (\mathsf{hJ}\ \mathsf{gx})$
= REST$_1$ J h $\mathsf{P_3}$

▷ Prove subresult $\mathsf{C_1Q_{4\text{-}1}}$: $\mathsf{gx} = (\mathsf{gJ}\ x)$
= REST$_1$ J g H_1

▷ Prove subresult $\mathsf{C_1Q_4}$: $(\mathsf{hJ}\ \mathsf{gx}) = (\mathsf{hJ}\ (\mathsf{gJ}\ x))$
= hJ.resp $\mathsf{C_1Q_{4\text{-}1}}$

▷ Prove subresult $\mathsf{C_1Q_5}$: $(\mathsf{hJ}\ (\mathsf{gJ}\ x)) = (\mathsf{hJ} \circ \mathsf{gJ}\ x)$
= (rewrite_compose hJ gJ x).symm

▷ Prove subresult $\mathsf{C_1}$: $(\mathsf{hgJ}\ x) = (\mathsf{hJ} \circ \mathsf{gJ}\ x)$
= ($\mathsf{C_1Q_1}$.tran $\mathsf{C_1Q_2}$).tran ($\mathsf{C_1Q_3}$.tran ($\mathsf{C_1Q_4}$.tran $\mathsf{C_1Q_5}$))

▷ Discharge H_1

▷ Suppose H_2 : $x \notin J$

▷ Prove subresult $\mathsf{C_2Q_1}$: $(\mathsf{hgJ}\ x) = x$
= REST$_2$ J hg H_2

▷ Prove subresult $\mathsf{C_2Q_2}$: $x = (\mathsf{hJ}\ x)$
= (REST$_2$ J h H_2).symm

▷ Prove subresult $\mathsf{C_2Q_{3\text{-}1}}$: $(\mathsf{gJ}\ x) = x$
= REST$_2$ J g H_2

▷ Prove subresult $\mathsf{C_2Q_3}$: $(\mathsf{hJ}\ x) = (\mathsf{hJ}\ (\mathsf{gJ}\ x))$
= hJ.resp $\mathsf{C_2Q_{3\text{-}1}}$.symm

▷ Prove subresult $\mathsf{C_2}$: $(\mathsf{hgJ}\ x) = (\mathsf{hJ} \circ \mathsf{gJ}\ x)$
= ($\mathsf{C_2Q_1}$.tran $\mathsf{C_2Q_2}$).tran ($\mathsf{C_2Q_3}$.tran $\mathsf{C_1Q_5}$)

▷ Discharge H_2

▷ Prove subresult P_4 : $((h \circ g)\restriction J\ x) = ((h\restriction J) \circ (g\restriction J)\ x)$
$= \mathsf{case}\ (x \in_? J)\ \mathsf{C}_1\ \mathsf{C}_2$

▷ Discharge x, h, g

▷ Prove $\mathsf{FT}_3\mathsf{iii}$: $\langle \exists \pi : \mathsf{iso}\ (K^\Delta/J^\Delta)\ (\mathsf{aut}\ J\ K)\rangle\ (\{\forall g : K^\Delta/J^\Delta\}\ (g\restriction J) = (\pi\ g)) \wedge$
$(\{\forall g, h : K^\Delta/J^\Delta\}\ ((h \circ g)\restriction J) = ((h\restriction J) \circ (g\restriction J)))$
$= (\pi, \mathsf{pair}\ \mathsf{P}_2\ \mathsf{P}_4 : \langle \exists \pi : \mathsf{iso}\ (K^\Delta/J^\Delta)\ (\mathsf{aut}\ J\ K)\rangle\ (\{\forall g : K^\Delta/J^\Delta\}\ (g\restriction J) = (\pi\ g))$
$\wedge\ (\{\forall g, h : K^\Delta/J^\Delta\}\ ((h \circ g)\restriction J) = ((h\restriction J) \circ (g\restriction J))))$

▷ Discharge $\mathsf{P}_4, \mathsf{C}_2, \mathsf{C}_2\mathsf{Q}_3, \mathsf{C}_2\mathsf{Q}_{3\text{–}1}, \mathsf{C}_2\mathsf{Q}_2, \mathsf{C}_2\mathsf{Q}_1, \mathsf{C}_1, \mathsf{C}_1\mathsf{Q}_5, \mathsf{C}_1\mathsf{Q}_4, \mathsf{C}_1\mathsf{Q}_{4\text{–}1}, \mathsf{C}_1\mathsf{Q}_3, \mathsf{P}_3,$
$\mathsf{P}_{3\text{–}6}, \mathsf{P}_{3\text{–}5}, \mathsf{P}_{3\text{–}4}, \mathsf{P}_{3\text{–}3}, \mathsf{C}_1\mathsf{Q}_{3\text{–}1}, \mathsf{C}_1\mathsf{Q}_2, \mathsf{C}_1\mathsf{Q}_1, \mathsf{hgJ}, \mathsf{hg}, \mathsf{hJ}, \mathsf{gJ}, \mathsf{hgx}, \mathsf{gx}, \mathsf{P}_2, \mathsf{Q}_4, \mathsf{Q}_3,$
$\mathsf{Q}_3\text{'}, \mathsf{Q}_{3\text{–}3}, \mathsf{Q}_{3\text{–}3\text{–}2}, \mathsf{Q}_{3\text{–}3\text{–}1}, \mathsf{Q}_{3\text{–}2}, \mathsf{Q}_{3\text{–}1}, \mathsf{P}_1, \mathsf{Q}_2, \mathsf{Q}_2\text{'}, \mathsf{conversion\_lemma}, \mathsf{Q}_{2\text{–}3},$
$\mathsf{Q}_{2\text{–}2}, \mathsf{Q}_{2\text{–}1}, \mathsf{g}_4, \mathsf{g}_3, \mathsf{g}_2, \mathsf{g}_1, g', \pi, \pi_4, \mathsf{P}_{4\text{–}1}, \pi_3, \mathsf{P}_{3\text{–}2}, \mathsf{P}_{3\text{–}1}, \pi_2, \pi_1, \mathsf{P}_{1\text{–}2}, \mathsf{P}_{1\text{–}1}, \mathsf{P}_{1\text{–}1\text{–}2},$
$\mathsf{P}_{1\text{–}1\text{–}2\text{–}1}, \mathsf{P}_{1\text{–}1\text{–}1}, \mathsf{apply\_GqJa}, \mathsf{GqJa}, G/U, G\restriction J, H, U, \mathsf{P}_0, \mathsf{Q}_{1\text{–}1}, \mathsf{G}$

▷ Discharge to prove as claimed FT_3
: $\{\forall L \mid \mathsf{dsubfield}\ F\}\ \{\forall K : \mathsf{galois\_subfield}\ L\}\ \{\forall J : K\text{-}\mathsf{subfield}\}$
$[\delta \mathsf{ft}_3\mathsf{A} = K.\mathsf{is\_galois\_subfield}\ J]\ [\delta \mathsf{ft}_3\mathsf{B} = J^\Delta.\mathsf{normal\_in}\ K^\Delta]\ (\mathsf{ft}_3\mathsf{A} \leftrightarrow \mathsf{ft}_3\mathsf{B}) \wedge$
$(\mathsf{ft}_3\mathsf{A} \rightarrow \langle \exists \pi : \mathsf{iso}\ (K^\Delta/J^\Delta)\ (\mathsf{aut}\ J\ K)\rangle\ (\{\forall g : K^\Delta/J^\Delta\}\ (g\restriction J) = (\pi\ g)) \wedge$
$(\{\forall g, h : K^\Delta/J^\Delta\}\ ((h \circ g)\restriction J) = ((h\restriction J) \circ (g\restriction J))))$
using
$\mathsf{pair}_3\text{'}\ \mathsf{FT}_3\mathsf{i}\ \mathsf{FT}_3\mathsf{ii}\ \mathsf{FT}_3\mathsf{iii} : \bigwedge_3 ((K.\mathsf{is\_galois\_subfield}\ J) \rightarrow J^\Delta.\mathsf{normal\_in}\ K^\Delta)$
$((J^\Delta.\mathsf{normal\_in}\ K^\Delta) \rightarrow K.\mathsf{is\_galois\_subfield}\ J)\ ((K.\mathsf{is\_galois\_subfield}\ J) \rightarrow$
$\langle \exists \pi : \mathsf{iso}\ (K^\Delta/J^\Delta)\ (\mathsf{aut}\ J\ K)\rangle\ (\{\forall g : K^\Delta/J^\Delta\}\ (g\restriction J) = (\pi\ g)) \wedge$
$(\{\forall g, h : K^\Delta/J^\Delta\}\ ((h \circ g)\restriction J) = ((h\restriction J) \circ (g\restriction J))))$

▷ Discharge J, K, L

▷ Discharge F

Index