Technical Report

UCAM-CL-TR-470
ISSN 1476-2986

Number 470

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Modular reasoning in Isabelle

Florian Kammüller

August 1999

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted April 1999 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Clare College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-470*

Abstract

This work is concerned with modules for higher order logic theorem provers, in particular Isabelle. Modules may be used to represent abstract mathematical structures. This is typical for applications in abstract algebra. In Chapter 1, we set out with the hypothesis that for an adequate representation of abstract structures we need modules that have a representation in the logic. We identify the aspects of locality and adequacy that are connected to the idea of modules in theorem provers.

In Chapter 2, we compare module systems of interactive theorem provers and their applicability to abstract algebra. Furthermore, we investigate a different family of proof systems based on type theory in Section 2.4.

We validate our hypothesis by performing a large case study in group theory: a mechanization of Sylow's theorem in Chapter 3.

Drawing from the experience gained by this large case study, we develop a concept of *locales* in Chapter 4 that captures local definitions, pretty printing syntax, and local assumptions. This concept is implemented and released with Isabelle version 98-1.

However, this concept is alone not sufficient to describe abstract structures. For example, structures like groups and rings need a more explicit representation as objects in the logic. A mechanization of dependent Σ-types and Π-types as typed sets in higher order logic is produced in Chapter 5 to represent structures adequately.

In Chapter 6, we test our results by applying the two concepts we developed in combination. First, we reconsider the Sylow case study. Furthermore, we demonstrate more algebraic examples. Factorization of groups, direct product of groups, and ring automorphisms are constructions that form themselves groups, which is formally proved. We also discuss the proof of the full version of Tarski's fixed point theorem. Finally, we consider how operations on modules can be realized by structures as dependent types. Locales are used in addition; we illustrate the reuse of theorems proved in a locale and the construction of a union of structures.

Acknowledgements

First and foremost, I want to thank my supervisor Lawrence C. Paulson. His sensitive guidance has enabled me to achieve more than I was hoping. When I was too reckless he advised care and when I was too frustrated he encouraged me. I have learned so much from him, maybe the most important lesson is: "keep it simple".

I am very grateful to my family and my friends in England and Germany. Without their support I wouldn't have had the energy to get through this. I don't want to name anyone because they all count individually and are equally important for me.

I want to thank Don Syme, Mark Staples, Reiner Kammüller, and Jacques Fleuriot for reading drafts of this thesis and giving valuable comments. The people in the Computer Lab inspired me in discussions and seminars. Thanks also to the anonymous referees that commented on papers about parts of my work and to all those people I had discussions with at conferences and other meetings.

Finally, I want to mention Mike Gordon and Tom Melham who gave me a wonderful viva and made it a delightful endpoint of my PhD and an encouragement for further work.

Contents

Chapter 1

Introduction

This thesis is concerned with modules for higher order logic theorem provers, in particular Isabelle [Pau94], and their application to the formalization of abstract algebra.

In computer science, modules are an element of programming languages. They are a tool to enhance the software engineering process. Modules support structuring, separate development and compilation of programs. Usually, they offer parameterization, which generalizes the code contained in a module, thereby enabling reuse. This is a form of abstraction.

Modules for theorem provers are introduced for similar reasons as in programming languages. In particular, generic theorem provers need some sort of modules to organize their object logics. These modules are often called *theories*. Like modules of programming languages, the ones for theorem provers are parameterized and hence introduce abstraction.

Abstraction in logic must be treated with care. Modules can represent structure. That is, if we represent a logical or mathematical structure by a module, then the module itself carries the meaning of this structure. Usually, modules are designed like their predecessors in programming languages as external devices, *i.e.* modules do not have any meaning in the logic of a theorem prover. Consequently, as soon as we employ modules to represent structures of an application, we cannot reason about these structures.

The abstraction that is an intrinsic part of modules for theorem provers is deeply related to mathematical and logical abstraction and is not just restricted to reusability. In order to emphasize the difference from programming languages, we use the terms *modular reasoning* and *modular structures* to describe the use of modules for mechanization of logic and the mathematical structures described by these modules.

Abstract algebra deals with abstraction in mathematics. It is an ideal application for the test of module systems for theorem provers. In comparison to most other abstract applications, like specification languages or other engineering methods for computer software and hardware development, it is

well understood semantically. Abstract algebraic structures typically consist of an abstract set and one or more abstract operations on this set obeying a few rules. This is why they may be represented elegantly by parameterized modules. Abstract algebraic proofs are often deep — properties *about* structures are used. On the other hand, abstract algebraic structures are characterized by only a few constructors and rules. Hence, they may be specified quickly. These features make abstract algebra a good test domain for the specification of structures in theorem provers.

Some people say that formalizations of mathematics, like mechanizations of abstract algebraic proofs, are not valuable for technological advance [QED96]. However, we believe that such formalizations may be useful to test module systems for theorem provers. Furthermore, abstract algebraic structures resemble elements of programming languages like objects and classes. Hence, it is very likely that module systems that enable a proper treatment of abstract algebra will improve formal support of software engineering as well. It is common in science that some subjects are critizised as just intellectual pleasures. In pure mathematics there are problems that seem of relatively little importance for any useful application in the real world. An example is Fermat's last theorem [Sin97]. This theorem is in itself not particularly interesting outside pure mathematics, the more so because it is a negative statement; it is concerned with the non-existence of solutions to an equation. Nevertheless, 358 years of attempts to solve Fermat's last theorem have led to many amazing developments in mathematics. New fields were explored, new techniques developed, and relationships between known domains of mathematics discovered. The actual solution connects modular forms with elliptic equations — two completely different domains of mathematics; a connection that was independently conjectured some 30 years before by Taniyama and Shimura but only solved by Wiles in order to prove Fermat's last theorem [Wil95].

We believe that formalization of mathematics plays a similar rôle in theorem proving. Attempts to construct good tools for the formalization of mathematics can produce techniques of general interest. As an example for this may serve the creation of higher order logic, which is the basis of many theorem provers that serve all kinds of practical tasks in software and hardware engineering. Higher order logic is usually traced back to Church's paper [Chu40], but its origin is to be seen in Russell's and Whitehead's work [WR62], which has been motivated by a crisis in the foundations of mathematics shortly characterized by Russell's paradox about the set of all sets.

Besides a contribution to the improvement of module systems for higher order logic theorem provers, this thesis should also be understood as an advocation of the formalization of mathematics. Presenting concepts that are suited to deal with the complicated world of abstract algebra, this work illustrates that general benefits for the theorem proving technology are gained.

Some issues we address are best described and explained here: throughout this thesis we will make use of a few principal notions that are specific to our area of research.

- Our utmost concern in this work is *adequacy*. By that we mean that the formal representation of a part of the real world in the language of a theorem prover should cover it as completely as possible. For the world of abstract algebra this means that it should be possible to find a representation for its constructs and to state propositions about these constructs (*logical* adequacy). These statements should at the same time be readable (*notational* adequacy). Furthermore, it should be possible — with reasonable investment of human and computer time — to conduct their proofs (*pragmatic* adequacy).

- Strongly connected to the notion of adequacy is the notion of *first class citizen*. By a first class citizen of a formal system we mean an element that lives in it. A first class citizen of a logic or a type theory is an entity that can be used in formulas. In higher order logic, first class citizens are those elements that may be denoted by a variable, *i.e.* are values. Alternatively, one may characterize them as elements that can be subject to a quanitfier. Normally, types are not first class citizens of higher order logic as they cannot be quantified (an extension that enables quantification over type variables is given by Melham [Mel93]).

- Another important aspect that comes with modules is best described as *locality*. Locality is the ability to declare concepts whose scope is limited or temporary. Locality for theorem provers enables the implicit use of information and thereby clarifies notations and facilitates proofs.

In the remainder of this chapter, we present short introductions to basic research fields that we will consider in order to tackle the task of constructing methods for modular reasoning in Isabelle. At first, we give in Section 1.1 a short introduction to type theory. We will use concepts from this theory in our developments. Although we will inspect type theory in more detail in Chapter 2, we consider it helpful to give a general introduction to it at the start. Abstract algebra deals with abstract sets and operations on them. Hence, one way to model abstract algebraic structures is to use polymorphic types, although mathematically the basis of an algebraic structure is a set. It is important to understand the differences between types and sets. We discuss in Section 1.2 the relationship between sets and types in a higher order logic theorem prover. Then we mention in Section 1.3 some work that has been performed towards a formalization of abstract algebra. Finally in Section 1.4, we develop our hypothesis stating a possible approach to a sensible treatment of modular structures for higher order logic theorem provers and outline the thesis in Section 1.5.

1.1 Type Theory

In the comparison of other provers, we come across a class of theorem proving assistants that are based on constructive type theory. They offer many interesting ideas, but unfortunately appear to be difficult to automate, for few large proofs have been undertaken using them.

It is important to understand that type theory is a fundamentally different approach to theorem proving. Although systems like Coq [D+93] and LEGO [LP92] do not differ much from a system like Isabelle or IMPS [FGT93] in the way they present themselves to the user, the underlying philosophy differs as much as the principal paradigm and hence the actual proof machine.

Whereas Isabelle and comparable systems are implementations of a logic, type theoretical provers are implementations of an abstract calculus of functions and types. The logic only comes in via the Curry-Howard isomorphism [How80]. This isomorphism is used as a paradigm saying that in the world of types we interpret every type as a proposition and terms inhabiting that type as proofs of that proposition. There is no explicit connection between the two worlds of types and truth *in* the type theory. The mechanisms of type theory, like reduction of terms, just happen to behave like some kind of logic and hence type theory can be used as a machinery for implementing logics. Since every term is constructed as something like a function term, type theory typically implements constructive logics.

1.2 Sets and Types in Higher Order Logic

Isabelle/HOL is a higher order logic theorem prover. Higher order logic is historically called a simple type theory [Chu40]. Types are used to impose hierarchies on the domains of terms. Types and terms are separate entities. There is one type, usually called *bool*, of all logical terms, *i.e.* all formulas.

Types do not play such a major rôle in higher order logic as they play in pure type theories. Still, types can be sometimes too restrictive to enable completeness of mathematical reasoning. Every set that is modelled by a type is restricted by the type hierarchy and the possibilities of type construction. Since the hierarchy of higher order logic is flat and the polymorphism is restricted to binding of type variables at the outermost position of the type, many higher order statements about a set represented by a type are not possible in such a model (*cf.* Section 5.4.2) .

Isabelle is a generic theorem prover. One can employ its different object logics for the reasoning process. Due to its untyped character the theory ZF, implementing Zermelo-Fraenkel set-theory, is the natural domain of mathematical reasoning. Since it is untyped, it is less convenient to represent structures in ZF. Abstract algebraic structures can profit from typing as it

is provided by higher order logic. The Isabelle object logic HOL, for higher order logic, offers a formalization of typed sets that we employ for abstract algebra. Nevertheless, a switch to Zermelo-Fraenkel set-theory could be a solution as well. Actually, a connection between HOL and ZF might offer expressive power of set theory combined with the convenience of type theoretic reasoning of HOL. To this end, one may construct an effective translation from HOL to ZF. One may use here the set theoretic semantics of HOL [GP93]. This interpretation is nontrivial because ZF has neither typed sets nor polymorphism. A mechanization of a theory translation mechanism for Isabelle is an interesting problem in its own right.

From a pragmatic point of view, this translation seems not to be necessary because the formalization of typed sets in HOL that we use turns out to be the best basis for our work. It combines convenience of types with expressiveness of sets.

Gordon experimented with an extension of the HOL system [GM93] with set theory [Gor95] to achieve more natural mathematical reasoning facilities. Polymorphism is represented there in terms of λ-abstraction with the set representing the polymorphic type as the function's parameter. For example, the polymorphic identity $\lambda\, x : a.\, x$ becomes $\lambda\, a.\, \{(x, x).\, x \in a\}$. In [Gor96] the author concludes that types as well as sets are needed. Lamport and Paulson discussing the use of types for specification languages [LP97] arrive at similar conclusions: generality of set theory can be combined with the benefit of type-based tools.

1.3 Formalizations of Abstract Algebra

Some work has been performed on applications of theorem proving to abstract algebra up till now. We mention only a few contributions that are milestones in this subject. Abstract algebra is a good domain for the study of expressivity of formal methods. The abstraction and structuring that is essential part of the reasoning process in abstract algebra challenges most formal languages and related tools. This is why we decided to use it as our field of application. The simple examples of groups, substructures of groups, like subgroups, and superstructures, like homomorphisms reveal very quickly weak points in formal systems or certain naïve characterizations of this part of mathematics.

The treatment of algebraic structures in the higher order logic theorem prover HOL has already been studied [Gun89]. However, at the time of these experiments there were no sets available in HOL. Hence, algebraic structures, like groups, are represented by a predicate over a base type. This formalization has some unnatural notational drawbacks. Gunter's formalization allows reasoning *about* groups because there are no explicit theory structures involved. Still, as far as logical adequacy is concerned, Gunter's

work can be seen as a benchmark for the present project.

Jackson has applied the type theory based system Nuprl [C+86] to abstract algebra [Jac95] and enhanced the system to deal with a computational subset of abstract algebra.

The case study [Bai98] of the formal proof of the fundamental theorem of Galois is a real milestone in the formalization of abstract algebra. This case study is performed in LEGO, a type theoretic proof checker offering automation for proof refinement. In this work, the author comments a few times on the problems with machine performance, which seem to have influenced the design decisions of the formalization. Apparently it had to be adjusted to achieve a reasonable performance at all. That should not be an issue when trying to mechanize mathematics. Although logically and notationally adequate, type theoretical mechanizations are not pragmatically adequate.

However, the concepts of type theory are expressive and elegant. The bad performance occurs because proofs are objects in the logic. Hence, the proof constructions have to be stored with the propositions. Naturally the proofs are magnitudes bigger than the propositions. An interesting approach would be to extract the elegant structures out of type theory without paying the high price for too rigorous principles. Can we achieve the logical and notational adequacy of type theory without sacrificing pragmatic adequacy?

1.4 Modular Reasoning

By *Modular Reasoning* we mean reasoning that employs modules as part of the logic. In a way this is a special case of what is often referred to as *abstract reasoning*. We restrict our attention to abstract algebra. For an adequate treatment of abstract algebra in an LCF-style higher order logic theorem prover, modules can be conveniently employed to represent algebraic structures. But, in that case, modules must be first class citizens of the logic. In this section we describe this observation in principle and discuss it. The observation sketched here will be further illustrated in Chapter 2.

1.4.1 Observation

To characterize the problem, we consider the example of groups that we will consider in detail in Section 2.3 and many other places in this thesis. In a *pseudo* notation for modules we can characterize groups as

```
Module Group [G: TYPE, o : G -> G -> G, inv: G -> G, e: G]
  ∀ x: G. x o e = x
  ∀ x: G. x o (inv x) = e
  ∀ x, y, z: G. x o (y o z) = (x o y) o z
```

This is the way modules are employed to characterize groups in systems like IMPS [FGT93], PVS [OSRSC98], and Larch [GH93].

With the use of modules the carrier of the group becomes a type. Thereby, the class of all groups is contained in the product type of the type of the carrier and the types of the other constituents of the group, *i.e.* the types of the binary operation, the inverse, and the unit element. The subtype of this product type that exactly represents the class of groups is then specified by the properties of the body of the module for groups.

To achieve adequate reasoning, we need to be able to express properties of all groups in the logic and use the class of all groups in the specification of substructures like subgroups or superstructures like homomorphisms or quotient groups. This is not possible if types and terms are on different levels, *i.e.* if formulas are terms while types are only a means to classify them. In higher order logic theorem provers, the latter is the case, *i.e.* terms and types are completely separate concepts.

Without considering types necessarily as logical properties as in type theory, we have to transfer some of the concepts of structure representation of type theory to logic to be able to grasp the world of abstract mathematics adequately.

The ideal solution would be to have on the one hand the convenient formalization employing modules and on the other hand to be able to change the viewpoint to consider these theory structures at the reasoning level. A notion of *reflection* between structures like modules and logical formulas seems to describe this change of viewpoint quite well. Reflection is a frequently used mechanism in reasoning processes [Har95]. In type theories the reflection is realized by impredicative types or to an extent by infinite type hierarchies, where each type can be considered as a term on a higher level (see Section 2.4). In IMPS we see that the solution to overcome the separation between terms, types, and theories (see Section 2.3.4) consists of transforming a sort to a set or indicator. This procedure is a reflection between types and terms.

Surely, reflection in the described sense cannot be introduced in general. Abandoning the separation between terms and types, we would encounter the foundational problems that were the reason to create types in the first place [WR62]. However, the applications of modules to abstract algebra we will present in Chapter 2 show that a proper treatment of this aspect in a structuring concept for a theorem prover is necessary to make reasoning more adequate. We aim to construct a support for modular reasoning that is more flexible with respect to reflection of structure. This will actually happen by separating the concerns of locality and adequacy.

1.4.2 Locality and Adequacy

One feature of a module is encapsulation. A module builds a context, or local scope, in which certain items are temporarily known. This feature is desirable, because is gives us *locality*. We can define local variables, constants, and operators in a notationally adequate way. These items are defined in a certain context. Thereby the context information can be used in their definition and enhances their concrete syntax. Locality is something that comes in with modules in the IMPS-PVS-Larch style.

The other aspect is *adequacy*. As we will see in Section 2.4, dependent types are well suited to give an adequate representation of algebraic structures. But, compared to the approach taken in type theory, we do not want to interpret these dependent types as logical constructors, instead just as structure representations. The approach of structures taken in the type theoretic proof tools solves the problem of adequacy, but the advantages of the modules of IMPS, PVS, and Larch in terms of locality are necessities for modular reasoning as well.

1.5 Overview

In Chapter 2 we compare the proof systems IMPS, PVS, and Larch and their module systems with respect to abstract algebra and point at the limitations. We also describe in more detail the class of type theory systems and their type universes.

The better theorem proving systems these days have grown into large formal frameworks. A system like Isabelle is in its generic concepts and different object logics so complex that it is hard to judge where the limits are, even for an expert user. Some limitations are quite obvious and yet there are often ways around them. The best way to find out if a system is suitable to fulfill the tasks it is constructed for — and at first sight seems capable of doing — is to apply the system to a large-scale case study. To that end, we continue this work with the application of Isabelle to a fundamental theorem of abstract algebra — the first theorem of Sylow. This case study was chosen because it deals with an application field that is not easily dealt with; the proof of this theorem outlined in [Her64] is a combination of several different ways of algebraic reasoning.

Much of the knowledge a first year student of abstract algebra learns is combined in this proof. For example, the proof uses Lagrange's theorem, which states that the order of a subgroup divides the order of the enclosing group. Furthermore, basic results about functions are used: to exhibit that Lagrange's theorem applies to a subset of the group, a bijection is defined between this subset and a factorization of the group. Besides reasoning about bijections and cardinality of sets, this requires reasoning about factorizations of a group. Finally, some combinatorial reasoning is developed

and applied to substructures to derive their properties. Summarizing, the proof demands a combination of different knowledge bases as well as viewing groups and substructures of groups under many different aspects. The variety of the involved reasoning entails the potential to reveal critical points of algebraic reasoning. In particular, using the convenient formalization in terms of modules performed in the provers IMPS, PVS, and Larch, even the formulation of the conjecture would be either difficult or impossible. The conjecture talks about cardinality and substructures of a group. To express such properties, a group has to be a first class citizen of the logic and it is not in the module characterization.

The case study yields a list of requirements that naturally leads on to the development of a concept of *locales* in Chapter 4. This concept addresses mostly the need for locality for the interactive proof process of higher order logic provers. We describe the concept and its integration into Isabelle. In addition, we sketch how the implementation has been performed. We discuss the possibility to reflect locales to achieve a first class representation for them. However, it turns out that such a representation is not easily achieved and furthermore not always desirable.

From that observation we continue the developments with the construction of structures as dependent types in Chapter 5. We describe our notion of algebraic structures, explain dependent types and their mechanization as typed sets in higher order logic, and describe how they may be used to represent structures.

Finally, we show in Chapter 6 how the concepts we have developed can be successfully combined to provide support for modular reasoning. First we reconsider the Sylow case study, but now apply the concepts of locales and structures as dependent types. We show the improvements in some detail and give some further insight into the conceptual enhancement. Following that, we consider more examples from abstract algebra validating the logical adequacy of the structural representation as dependent types. We formalize the factorization of groups, direct product of groups, the group of ring automorphisms, and the full version of Tarski's fixed point theorem. Locales can be used in perfect interaction with the first class representation by dependent types thereby smoothly merging adequacy with locality.

Chapter 7 concludes the work with a summary and points out the achievements. It also discusses some lessons we learned during the project and sketches a few possibilities for future research.

Chapter 2

Modules

We begin the work in Section 2.1 with a short introduction to the theorem prover Isabelle and the attempts that have been made to build modules for Isabelle. Isabelle did not have a module concept prior to the current development, but there has been a proposal [Pau91b] and experiments [Asp91]. In addition, Isabelle's polymorphism gives rise to a notion of axiomatic type classes [Nip93, Wen95] that is integrated in the current release. This concept can be seen as a particular case of modules (see Section 2.1.3).

A good starting point to explore the possibilities for a support of abstract reasoning and modularization in Isabelle is to look at module concepts of comparable proof systems. We will point out requirements for the development of modular structures for algebraic reasoning by comparing the module concepts of IMPS, PVS, and Larch in Section 2.2. We discuss in Section 2.3 how the developers of these provers suggest to employ their modules for abstract algebra. Theorem provers like Coq and LEGO are in a different class of theorem proving systems because they use type theory. They do not have module systems comparable to the former provers, but they have other interesting concepts for the formalization of abstract algebra. They will be considered separately in Section 2.4. This chapter wants to validate the hypothesis we put forth in Section 1.4 by comparing different modular concepts and their usability for abstract algebraic proof.

2.1 Isabelle

Isabelle [Pau94] can be instantiated to form theorem provers for a wide range of logics [Pau90]. Thus, it is well suited for the development and testing of new logics. These can be made known to the prover by defining theories that contain sort and type declarations, constants, and related definitions and rules. A powerful parser supports intelligible syntactic abbreviations for user-defined constants.

Apart from being a tool for logical developments, some instantiations

of Isabelle have independent value as theorem provers. These are Zermelo-Fraenkel set theory (ZF), higher order logic (HOL), and constructive type theory (CTT). The best developed and most widely used ones are ZF and HOL. Substantial case studies have been performed in both of them (*e.g.* [Pau95, Pau98]).

Isabelle's meta-logic is a fragment of higher order logic [Pau89]. Some parts that are missing are existential quantification and conjunction. The type system of Isabelle is a simple type theory with Hindley-Milner polymorphism. It is a traditional higher order logic in that it has no dependent types or subtypes as some variations of higher order logic do (see Section 2.2). Nevertheless, Isabelle/HOL has a device to define new types through set definitions over already existing types. This is a kind of subtyping but it is rather like modelling subtypes using sets and isomorphisms. For example, these new types are completely separate from their supertypes. That is, there is no automatic or implicit coercion and functions do not carry over from supertypes to subtypes. The concept of axiomatic type classes integrates some advantages of the subtype concept on the more abstract level of type classes.

In the subsequent section we briefly explain Isabelle's notion of theory.

2.1.1 Isabelle Theories

An Isabelle theory contains all axioms, definitions and other kinds of declarations, like for example type definitions. Object logics of Isabelle, like HOL and ZF, but also ones defined by users, reside in such theories. New theories can be built from existing ones *via* extension.

To allow axiomatizations of object logics, the rules of object logics can be assumed to be true without proof. The responsibility is with the user to ensure that the set of rules is sound and consistent with underlying logics. Nevertheless, conservative extension may be built if one uses only definitions. For higher order logic the nonemptiness of a new type is a necessary requirement to sustain consistency. In the object logic HOL of Isabelle this is true as well if the new types are constructed using `typedef`. Then the nonemptiness condition is produced as a proof obligation.

Definitions, rules, and other declarations that are contained in an Isabelle theory are visible whenever that theory is loaded into an Isabelle session. All theories on which the current theory is built are also visible. All entities contained in a current theory stay visible for any other theory that uses the current one. Thus, Isabelle theories form hierarchies. But, theories do not have any parameters or other advanced features typical for modules in theorem provers (see Section 2.2). We introduce next a concept of Isabelle Modules as it has been proposed by Paulson [Pau91b] and implemented by Aspinall [Asp91].

2.1.2 Isabelle Modules (1991)

Isabelle Modules are designed according to the ML concepts [Pau91a] of structure, signature and functor. But, signatures can also include axioms. These axioms have to be fulfilled in structures which match the signature. Functors map structures to each other in order to define new ones from others. We first introduce in some detail these notions of signature, structure and functor and then sketch the definition of context.

Structures

Structures build the main object of interest in the module concept. They are classified by signatures and transformed by functors. A structure consists of bindings which introduce abbreviations for types or names for existing structures and definitions, which associate constants with terms.

Theorems are proved "inside" a structure; they may be associated with names and are recorded in the structure. The components of a structure are not directly visible from the outside, but may be referred to by compound names, *e.g.* *Str.c* for component *c* of structure *Str*. Since theorems are stored in it, an Isabelle structure involves some state. The Isabelle theories of the current version of Isabelle are what structures in the concept of Isabelle Modules were at the time: non parametric entities containing constants, types, rules, and definitions.

Signatures

Isabelle Modules differ from ML modules in that in Isabelle a signature may include axioms that state requirements on structures, which *match* this signature.

Signatures define classes of structures. They specify types, constants with types, and substructures with their signature. Furthermore, they specify concrete syntax via mixfix declarations and names for the axioms. By specifying substructures as *open* in a signature all the items of the opened structure are added to the signature and made visible whenever an instance of the signature is opened.

To specify that two substructures are the same, a signature may contain sharing constraints. "The same" is an extensional notion here. It means equality rather than identity, unlike in ML. ML has no structural equality, *i.e.* if we define an individual structure twice with different names the structures are not identical.

There are two levels by which a structure can match a signature. A structure can *match* and *satisfy* a signature:

- a structure *Str* *matches* a signature *Sig*, written *Str*$*Sig*, if *Str*
  - binds all the types defined in *Sig*

  - defines all constants specified in *Sig*, giving them the specified types
  - binds all the substructures specified in *Sig*, giving them the specified signatures.
  - satisfies all the sharing constraints in *Sig*.

- a structure *Str satisfies* a signature *Sig*, written *Str* : *Sig*, if *Str* matches *Sig* and additionally all the axioms in *Sig* have been proved as theorems in *Str*.

Matching can be tested automatically because it is just a type checking operation whereas satisfaction may involve proofs, which have to be performed by the user. Since a foundational system like first-order logic or set theory cannot be implemented starting from nothing, it is assumed for a given signature that some structure satisfies it. This is done by declaring a structure as *primitive* for a particular signature.

Functors

A functor provides a method for building a structure from other structures. It takes the structures as formal parameters and defines another structure from those. The functor may be applied to structures which match its formal parameters. It produces instances of its result signature. Thereby, proof obligations of the result signature may be created. These can be solved by theorem proving in the body of the functor. After that the functor is said to be *promoted*: its application yields a structure satisfying the result signature. If the obligations are not solved the application results in structures which just match the result signature.

Contexts

A context is an environment, which is used to define module elements. In the definition of context Paulson [Pau91b] and Aspinall [Asp91] differ. They agree about a *global context*, though. This shall be the "current theory" like in HOL or LCF. This global context contains the primitive structures and all signatures and functors are defined within. The structures in the global context are *pervasive* — they are visible everywhere.

Aspinall [Asp91] defines a notion of *current theory*, which is formed by adding the contents of structures to the global context. This addition can be undone. So, a kind of temporary opening facility is provided. The proposal [Pau91b] says that the global context contains all structures that have been previously created within the global context. But there a notion of *local* context is defined in terms of functors: each functor heading defines a local context which contains the functor's formal parameter and also the global context.

The global context avoids the need to specify pervasive items in signatures or as formal parameters to functors. If structures could only be mentioned as formal parameters, then their properties would have to be imported into each functor and furthermore declared as sharing.

In general, the functor concept is obviously able to perform all typical actions of the module systems of PVS, LSL, and IMPS, which we are going to consider in Section 2.2. Yet, it also suffers the same restrictions that we describe in Section 2.3. Modules cannot be Isabelle values since they may contain a type and we cannot build formulas over types in Isabelle's meta-logic.

Isabelle Modules have not made their way into the current release. In the present state of Isabelle, a theory is something like a primitive structure satisfying a given signature. However, there is another feature of the Isabelle system that supports modular ideas. This is the concept of axiomatic type classes.

2.1.3 Axiomatic Type Classes

The type class system of Isabelle [Nip93] enables the organization of the polymorphic types of object logics in a lattice-like order. By using type variables and coercing them into certain classes it becomes possible to state axioms for whole classes of types. If a type is declared as a member of the type class, then all axioms are inherited. The mechanism of defining type classes together with axioms is an explicit device of the Isabelle theory definition language [Wen95].

A type class in Isabelle is a device for grouping types. For example, `semigroup < term` defines a type class as a subclass of the built-in class `term` of HOL. Type classes can be defined directly together with axioms, which members of the class have to fulfill.

The concept of these classes is best illustrated by an example.

```
Semigroup = HOL +
  consts
    "*"           :: "['a, 'a] => 'a"     (infixl 70)
  defs
    assoc_def
      "assoc f == ALL x y z. f (f x y) z = f x (f y z)
  axclass
    semigroup < term
    mul_assoc     assoc(op *)
  end
```

After declaring an operation `*` over a type class, the `axclass` section allows us to define a new subclass `semigroup` of the standard type class `term` of HOL. The axiom `mul_assoc` is then valid in the whole type class `semigroup`,

i.e. for all types which will be assigned to this class and all types which will be assigned to future subclasses of `semigroup`.

Besides the possibility to define a class of structures meeting abstract conditions prescribed in the `axclass` section, it is also possible to *instantiate* `axclasses`. Assuming an axclass definition of groups and monoids, this is illustrated by:

```
MonoidGroupInsts = Monoid + Group +
  instance
    monoid < semigroup          (Monoid.assoc)
  instance
    group < monoid              (assoc, left_unit, right_unit)
end
```

The instance sections set up the necessary obligations for the class inclusions. Corresponding goals are produced automatically and their proofs attempted using theorems supplied by the user in brackets on the right side. This form of annotation advises Isabelle to solve the produced obligations by the listed properties. Thus, we can insert the class `monoid` between group and semigroup *wrt* <. The instance construction allows whole classes of structures to be put into an axiomatic class order. Note that the `instance` facility performs a restricted action of a functor as described in Section 2.1.2. That is, a type class together with its axioms may be seen as a signature with an abstract sort. The instance of this signature to another signature, *i.e.* type class, is equivalent to the definition of a functor mapping the first signature to a result signature. An instance of a type into this type class lattice corresponds to the application of the functor to structures. In the above case `group < monoid` maps from groups to monoids, *i.e.* every group is a monoid. The inclusion obligations can be proved on an abstract level thus becoming simply applicable to concrete types.

Extension of Type Classes

If we think about modular structures in Isabelle, we might consider the axiomatic type classes as a starting point. They represent a particular case of modules. An axiomatic type class defines a type class with certain properties. As we have seen, the instance section

```
instance
  monoid < semigroup          (Monoid.assoc)
```

for the definition of the `axclass monoid` enables inserting one type class into another. This corresponds to a theory interpretation or an import in other theorem provers (see Section 2.2).

The idea arises to express the functor concept of Isabelle-91 in terms of `axclasses`. However, the correspondence works only for cases of simple type signatures. Unfortunately, the `axclass` concept is restricted to one

type [Nip93]. To achieve a mechanism of abstract theories similar to other theorem provers in terms of `axclasses` it would be necessary to allow more than one abstract type in an `axclass`. We need to extend the lattice of the type classes to whole *type signatures*. Consequently, as soon as we want to express signatures (and functors) over different base types the current concept of `axclasses` is not sufficient any more.

Also, the structures expressed as `axclasses` are not first class citizens of the logic, *i.e.* they are not terms. We are not able to express properties where the structures themselves are objects of formulas. Explicit reasoning about structures, *i.e.* axiomatic type classes, is not possible. As in the module concepts of other provers, the type classes are on a different level from the terms of the reasoning (*cf.* Section 2.2).

For the above `instance` section, Isabelle produces the proof obligation `OFCLASS(monoid,semigroup)`. This obligation could be considered as a predicate stating *a monoid is a semigroup* thus representing a particular case of a reflection of the structures semigroup and monoid onto the reasoning level. Unfortunately, the `OFCLASS` obligation is an internal function which is — although a visible obligation — not accessible by the user, *i.e.* it cannot be manipulated or used explicitly on the reasoning level. Allowing access to this internal `OFCLASS` predicate would at least admit reasoning about a particular predicate of structures.

Summarizing, an extension of the concept of axiomatic type classes is a possibility to establish modular structures in Isabelle. However, we believe that the construction of something like *axiomatic type signatures* demands a lot of theoretical preparation. The extension of axiomatic type classes to more than one type makes the type checking process more difficult, if not undecidable.

2.2 Modules in IMPS, PVS, and Larch

There are many proof systems which attempt to be generic in the sense that they provide support for various tasks, *i.e.* they may be instantiated to different problem worlds and offer their adjusted proof power there. Among them are HOL, Isabelle, PVS, IMPS, Eves, LEGO, and Coq. Among the known systems, some offer possibilities to support modularization and parametrization of theories. We chose IMPS [FGT93], PVS [OSRSC98], and Larch [GH93] as adequate candidates for a comparison because they have an explicit notion of theory. Their module systems appear to be successful and elaborate.

The HOL system [GM93] has got a theory device for its version 98 [Sli98]. This is designed according to the module system of Moscow ML, *i.e.* HOL developments reside in ML structures[1]. It does not contain an explicit notion

[1] Moscow ML does not have functors.

of theory in the specification language of the theorem prover. Theories are represented by separately compiled ML structures. In contrast to our main objective of adequacy, HOL theories address the efficient organization of proof developments. Earlier work on *abstract theories* [Win93] is more along our lines (see Section 5.4).

Another possible example for a comparison was Eves [Saa89, KPS+92, SC91]. However, after having studied the relevant publications we came to the conclusion that its facilities of theory construction do not offer anything remarkable. Though there is a device to relate specifications to models in a mechanical way in the Eves library [SC91] the specification language of Eves, Verdi, has none of the constructions present in the other systems.

After short introductions to the three systems IMPS, PVS, and Larch, we will describe their module systems in more detail.

2.2.1 IMPS

The Interactive Mathematical Proof System (IMPS) [FGT92a, FGT93] developed at MITRE is mainly designed for interactive machine supported mathematical proof. It emphasizes the linking of axiomatic theories as the main method of mathematical reasoning [FGT92b]. The heart of IMPS is its higher order logic LUTINS, a Logic of Undefined Terms for Inference in a Natural Style. The speciality of LUTINS compared to other logics based on simple type theory is its explicit notion of partial functions.

The type hierarchy of LUTINS consists of base types and function types. For a language $\mathcal{L}$ in LUTINS there is always the type of propositions `prop`, and depending on $\mathcal{L}$ some types of individuals. Function types are defined over those atomic types as $\alpha_1, \ldots, \alpha_n \to \alpha_{n+1}$ for base or function types α_i and $i \in \{1, \ldots, n+1\}$.

The terms of `prop` are always defined and hence all predicates are total functions. Arbitrary functions may be undefined for certain inputs. IMPS provides the kind *ind* of individuals, which is the set of all types of possibly partial functions. Except for undefined terms, all terms are typable, *i.e.* LUTINS is strongly typed. On top of the type hierarchy there is a sort system. Types and subtypes together build the sorts of this sort system.

LUTINS contains no polymorphism in the sense of variables over types. In [FGT93, p.219] the authors say that polymorphism is achieved by the use of constructors, quasi-constructors, sorts and theory-interpretations. Quasi-constructors are basically the same as constructors; one reason to have them is that the user cannot define constructors. Constructors and quasi-constructors are globally valid constants, *i.e.* they can be used across theories.

The philosophy of IMPS may be shortly characterized by "Little Theories" [FGT92b]. Theories can be seen as modules. Mathematical reasoning in IMPS is as well a matter of relating theories or transporting theorems

from one theory to another as reasoning in one large theory, like Zermelo-Fraenkel set theory. Theories and their contents are viewed as objects to be linked, transported, and interpreted. There are some specific methods to realize this. To be able to analyze the possibilities of the IMPS module system we concentrate on the central aspects.

Definition of Theories

A theory is constructed from a possibly empty set of *subtheories*, a language, and a set of axioms. Theories may be enriched at any time by the definition of new atomic sorts and constants and by the *installation* of theorems. The relations on theories are subtheory and theory interpretation. Theories and copies of theories can be grouped together in so-called *theory ensembles.*

Theory development in IMPS is a dynamic process. The user defines and develops a theory, theory interpretation, or other objects by evaluating expressions called definition forms (or *def*-forms, for short). There are some 30 different def-forms in IMPS. The IMPS system relies basically on the global environment or context in which several different theories "live" and undergo changes during a session.

Theory Interpretation

The central mechanism of IMPS is theory interpretation [FGT92b]. The notion of theory interpretation in LUTINS is very similar to the standard notion in first-order logic apart from complications caused by the partiality. Intuitively, a theory interpretation from a source theory $\mathcal{S}$ to a target theory $\mathcal{T}$ specifies one way of embedding $\mathcal{S}$ in $\mathcal{T}$, while preserving theorems. This notion is formalized using certain syntactic translations. In brief, a LUTINS translation from a theory $\mathcal{S}$ to $\mathcal{T}$ is specified as a pair (μ, ν) of functions. The function μ maps the sorts of $\mathcal{S}$ to sorts, sets or unary predicates of $\mathcal{T}$ and ν maps the constants of $\mathcal{S}$ to expressions of $\mathcal{T}$. This translation is a kind of homomorphism Φ from the expressions of $\mathcal{S}$ to the expressions of $\mathcal{T}$, *i.e.* $\Phi(c(e_1, \ldots, e_n)) = \Phi(c)(\Phi(e_1), \ldots, \Phi(e_n))$, where c is a constant and $e_1, \ldots e_n$ are subexpressions.

Every translation Φ determines a set of formulas in $\mathcal{T}$, so-called *obligations.* These obligations contain in particular the Φ-images of all axioms $\mathcal{S}$. If each obligation of the source theory $\mathcal{S}$ is a theorem of the target theory $\mathcal{T}$ then Φ is a theory interpretation by the interpretation theorem of LUTINS [Far93]. That is, Φ translates each theorem of $\mathcal{S}$ into a theorem of $\mathcal{T}$.

Interpretations are independent from languages $\mathcal{L}$, so they form a means to relate arbitrary user defined applications. If there is a theory interpretation in IMPS from a theory $\mathcal{S}$ to a theory $\mathcal{T}$, then $\mathcal{S}$ is consistent if $\mathcal{T}$ is consistent. Thus, theory interpretations provide a mechanism for showing that one theory is consistent relative to another.

Syntactically, theory translations are handled in the same manner as definitions using def-forms. That is, in one command language the definition of theories itself as well as their relations and interpretations are expressed. For example, the def-form that defines the symmetry translation which reverses the group[2] multiplication, reads:

```
(def-translation MUL-REVERSE
  (source groups)
  (target groups)
  (fixed-theories h-o-real-arithmetic)
  (constant-pairs
      (mul "lambda (x, y: gg, y mul x)"))
  (theory-interpretation-check using-simplification))
```

The system uses the simplifier to check that the theory interpretation obligations of this translation hold. Then the translation is an interpretation.

The function μ of the above theory translation is $\mu(\alpha) = \alpha$ for α the base sort of elements of the group, and μ(prop) = prop for the type prop. The second part ν maps every constant to itself except $v(mul_{[\alpha,\alpha,\alpha]}) = \lambda x_\alpha, y_\alpha . mul_{[\alpha,\alpha,\alpha]}(y_\alpha, x_\alpha)$. The notion of arity $[\alpha, \alpha, \alpha]$ stands for the compound sort $\alpha, \alpha \to \alpha$.

Generic Theories

In this section we introduce the notion of generic theories in IMPS in two ways. First we give an account on how abstraction is achieved *via* modules in IMPS, and then we explain the definition of "generic theories" as the designers of IMPS explain this term.

There is no explicit notion of parameterization in theory interpretations of IMPS. Interpretations are constructed by mapping a usual theory to another one by defining a translation. There are no parameters at all; basically any concrete element that is mapped can be viewed as an actual parameter to a (non-existent) formal parameter. Thus, every theory which is *mathematically* abstract can be regarded as generic. By defining a theory translation from this "generic" theory to any specialization we introduce its abstract character. The abstraction can be developed step by step and need not be stated explicitly when defining a theory. This absence of explicit polymorphism is a decisive difference between IMPS and a system like Isabelle whose universal polymorphism is explicitly realized with type variables.

The explicit notion of "generic" theories in IMPS is used for theories containing neither axioms nor constants. Theories of generic types are defined in the IMPS documentation [FGT95] as theories with nothing in them except a base type. The user is advised to prove results about quasi-constructors which are built merely on logical constants in those generic theories because then these results are derived in the most general way. Any use of

[2]The definition of the theory will be given in Section 2.3.1.

these results in other theories is easily possible because they may be trivially transported there by theory interpretation. An example of such a theory is the theory for sets. Those are represented by characteristic functions called *indicators*. An indicator in IMPS is a function which takes on a fixed value — say 1 — on elements which "belong" to the indicator, *i.e.* are members of a set, and which is undefined for all other arguments. Such indicator theories are generic in the sense that their theorems can be easily transported to other theories because they contain neither axioms which would produce obligations nor explicit IMPS sorts which needed to be mapped somehow. The sort of an indicator is `[`α`,unit%sort]` where α is an arbitrary sort[3] and `unit%sort` is a type of a kernel theory which is part of every IMPS theory and contains only one element called `an%individual`. The indicator is printed as `sets[`α`]`.

2.2.2 PVS

The Prototype Verification System (PVS) developed at SRI International is designed for the development and analysis of formal specifications. The PVS system has applications in this field [Rus92, SM95, MS95]. The PVS specification language builds on what they call classical typed higher order logic.

A PVS specification consists of a collection of theories constituted by a signature, axioms, definitions and theorems. Entities of PVS are mainly introduced by means of *declarations*. Declarations are used to introduce types, variables, constants and formulas. Each declaration has an *identifier* and belongs to a unique theory. Declarations also have a body which indicates the *kind* of the declaration and provides the signature and definition of the declaration. Declarations are classified according to their kind; these kinds are *type*, *prop*, *expr*, and *theory*. Expressions, *i.e.* items of kind *expr*, can be again variables, constants, and recursive definitions.

Functions in PVS are all total. Partial functions may be modelled by defining them as total functions over subtypes.

PVS has a strongly typed specification language, *i.e.* every expression is typable, although not automatically because type checking is undecidable. The PVS specification language contains *type constructors* for building more complex types, like the function type `[int -> int]`. The built-in set of type constructors contains constructors for subtypes, function types, predicate types, tuple types, and record types. The function, tuple and record types may be *dependent* types, *i.e.* in the declaration of a type later components may depend on earlier components of the same type declaration.

PVS offers *predicate* subtypes which may be defined in a set-like notation, *e.g.* `t: TYPE = {x: S | p(x)}` where `p` is the predicate defining the

[3]The name α seems to indicate a type variable, but this is not the case. However, it can be considered as abstract by interpretation as described in the previous paragraph.

subtype. This form of defining subtypes provides expressivity but leads to undecidable type checking; type checking conditions, so-called TCC's, are generated in undecidable cases. They have to be solved by the user.

The PVS Tutorial [OSRSC98, p.55] claims that the theory parameterization provides support for universal polymorphism. This is not explicit polymorphism in the sense of type variables, but achieved rather by parameterization of theories.

Definition of Theories

In PVS, the user defines theories in a declarative style. The syntactical form of a theory looks much like an imperative programming language module. There are fixed sections of contents in a theory, *e.g.* it has an assumption part and an `EXPORTING` section.

A PVS theory consists of a theory identifier, a list of formal parameters, an `EXPORTING` clause, an assuming part, and a theory body. In the PVS system, the set of theories available in a session forms a *context* in which theory names must be unique. An initial context, called "prelude", provides convenient entities like the Boolean operators, equality, and the real, rational, integer, and natural number types and their associated properties. This prelude is automatically imported into every theory.

The *assuming part* of a theory precedes its theory part. Intuitively, this assuming part states constraints on the use of the theory. These constraints are defined in form of formulas preceded by the keyword `ASSUMPTION`. The assuming part can also contain declarations needed in its formulas and imports. Apart from the variables, these declarations are visible from the outside as well. The constraints defined in the form of assumptions have to hold for any instance of the theory and then become obligations which must be discharged. Internally, assumptions may be used as axioms; externally, they have to be proved for each import of the theory with the actual parameters replacing the formal ones.

The *theory part* usually contains the main body of the theory. It consists of top level declarations and `IMPORTING`s. The declarations may be axioms, labeled by the keyword `AXIOMS`. Furthermore, theorems may be stated by `THEOREM` but there are also several other keywords, *e.g.* `LEMMA`, which all mean the same and serve for "greater diversity in classifying formulas". Free variables are possible and handled by the universal closure. Assumptions are not allowed in the theory part.

Theory Hierarchies

PVS theories may make use of other theories. They may export parts of their entities to let others use them. This import-export relation between theories reminds us again of imperative programming languages like Modula-2. The

relation must form a hierarchy.

- `EXPORTING`:

  Some names declared in a theory may be hidden from other theories whereas others may be made visible in the same context. This is achieved by means of the `EXPORTING` clause. Names exported by a theory may be imported into a second theory. Exporting is an option in PVS: if omitted it defaults to exporting everything. The advice of the authors is "it is probably best not to include an exporting clause unless there is a good reason" [OSRSC98, p.57]. An exporting clause may specify instances of the theories, which the current theory used, to be exported.

- `IMPORTING`:

  Visible names of other theories in the current context may be imported into a theory. The `IMPORTING` clause can appear in the parameter list, the assuming part, or the theory part of a theory. An import with actual parameters provided is called a *theory instance*, even if it has no parameters. An import without instantiation of parameters is called a *generic reference*. Importing forms a relation between the theory referenced and the theory which imports. The transitive closure of this import relation is called the *importing chain* of a theory. The importing chain must form a directed acyclic graph, so that a theory may not import itself, directly or indirectly.

We see that the theory hierarchies of PVS are inferior to the notion of translation in IMPS that enables self-reference as is illustrated by the example `MUL-reverse` in Section 2.2.1. PVS cannot have self-reference because that would correspond to a theory importing itself. Self-reference of theories can be very useful as it enables proofs using duality.

Theory Parameters and Instantiation

PVS theories may be parametric in types and values, as specified by the parameter list in square brackets following the theory identifier. More precisely, these parameters may be types, subtypes, constants, or imports. Apart from the constraints defined in the specification of the parameter list, the parameters may further be constrained by the assumption part of the theory. The parameter concept provides a certain kind of polymorphism named "universal polymorphism" in [OSRSC98]. This is comparable to the polymorphism in IMPS although it is more explicit. Inside the parameter list, later parameters may refer to earlier ones, whereby a dependence of types may be placed as a constraint on parameters.

A theory is instantiated in PVS from within another theory by inserting actual parameters for the formals. This insertion is called a *theory*

instance. A theory instance may be given a name by which this instance and its constituents may then be referred to. For example, the code fragment `fsets: THEORY = sets[[integer -> integer]]` names the instantiation of the theory of sets by integer functions as `fsets`. If the theory `sets` contains a predicate `member` this element of the instance may be referred to as `fsets.member`. Although this notation seems to indicate that theories are first class citizens, it only enables references to contents of theories.

2.2.3 Larch

Larch [GH93] is a system consisting of several axiomatic specification languages. One group of languages is designed for the specification of interfaces between program components. Such *interface languages* exist for C (LCL [GH91]) and Modula-3 (LM3 [Nel91]), for example. There is the language LSL, the *Larch Shared Language* which all interface languages have in common. It is independent from any programming language and intended to specify the meaning of the languages given in the interface specifications. The *Larch Prover*, LP, serves as a proof assistant to reason about Larch specifications or to assist in "specification debugging" [GGH90]. In contrast to the specification languages of PVS and IMPS, LSL is based on typed *first-order* logic with equations. An LSL specification declares in its **introduces** section a list of *operators* given by function identifiers and their signature. The signature is a characterization of the operators domain and range, *e.g.* `__ ∈ __ : Ind, Tab → Bool` declares an operator $\in$ as a function taking an argument of sort `Ind` and one of sort `Tab` mapping them to sort `Bool`. The signatures allow Larch to check the sorts of terms, which is comparable to type checking. The **introduces** section is part of a so-called *trait* which builds LSL's basic unit of specification. The remainder of a trait is constituted by a section indicated by the keyword **asserts**. This section defines properties of the operators by means of equations. Further features of basic character include various built-in Boolean operators and syntactic shorthands like *enumerations*, *tuples* and *unions* which provide representations for common structures.

Sorts in LSL are represented by symbols. A sort is a non-empty set of objects. Distinct sorts represent disjoint sets of objects. The only built-in sort is `Bool`, which represents a set containing the objects `true` and `false`. Compound and function sorts are built by grouping sorts in tuples or using the constructor symbol $\rightarrow$. There is no explicit notion of subsort or inheritance of operators for subsorts.

There are two constructs in LSL for determining the structure of sorts more precisely. The clause **generated by** followed by a list of operators over a sort specifies that this sort can be generated by the enumerated operators. A **partitioned by** clause allows one to identify terms of sorts which cannot be distinguished by a given list of operators.

As already mentioned above, the *trait* is the basic unit of specification in LSL. It is meant to contain an abstract data type or possibly encapsulate a property which is common for several data types. Although it seems to be rather a unit in the sense of a programming language module or object, its logical meaning is that of a theory: "each trait defines a theory (a set of formulas without free variables) in typed first-order logic with equality" [GHM90]. A trait is constituted by its **introduces** part, the section declaring all the operators of the trait, and its **asserts** part, which contains the constraints on the operators in form of equations and possibly the **generated by** or **partitioned by** clauses. A further part of a trait is the *implies* section, which allows one to state claims about "theory containment", *i.e.* theorems representing consequences of the traits assertions. In this section, a list of formulas may be stated which have to be shown. It is noteworthy that in the **implies** section the full power of the language may be used which includes equations, **generated by** and **partitioned by** clauses. Furthermore, universal quantification may be used as well as references to other traits. For example, let `Associative` be the following trait:

```
Associative(∘, T): trait
introduces __ ∘ __ :  T, T → T
asserts ∀ x, y, z :  T
(x ∘ y) ∘ z == x ∘ (y ∘ z)
```

Another trait which introduces a binary associative operator `op` over a sort `T'` may contain `Associative(op, T')` in its **implies** section. The insertion of this formula in the **implies** list states that it should be shown that `op` is indeed associative.

Parameterization of Traits

As in the trait example `Associative`, it is possible to use parameters to define general specifications. Parameters are just listed after the trait's name without specifying them in any way. They may be sorts or operator symbols. General specifications may be "specialized", *i.e.* another trait may load an instance of a general trait. For example, if `Set` is a trait with parameter `S` abstracting over the element type another trait could use the code **includes** `Integer, Set(Int)` to instantiate the general `Set` trait to integer sets. Thereby, the included trait becomes accessible in the present one with the actual parameter inserted for the formal one.

Another option of the trait structure is to pose requirements on the parameters. By the keyword **assumes** positioned after the trait header an optional section may be opened for stating assumptions on the formal parameters. The **assumes** formulas are restricted to references to other traits possibly with actual parameter instantiation.

Renaming of Trait Contents

Besides instantiation, sorts or operators may also be just renamed. That is, whenever a trait is included in another, the sorts and operators of the incorporated trait can be renamed using the **for** clause. The formula part (*newname* **for** *oldname*) can be inserted after **includes** *trait* and results in the replacement of every occurrence of *oldname* by *newname* in the entire trait. This device does not exist in the other systems. If a trait includes different other traits which define properties of an equally named operator their union automatically identifies the operator by its name and unifies its diverse properties. Still every name has to be declared in the **asserts** part or must be made present by an **includes** declaration (possibly with renaming) if this name is used in a trait. Names do not have global existence; they are visible only in traits. Thus, we have a kind of scoping, *i.e.* names are only visible in a trait if made known in some way.

2.2.4 Comparison

Summarizing, the exposition of PVS, IMPS and Larch shows that Larch has some nice ideas but cannot compete with the two others. For example, Larch theories can be used *quasi* as formulas at the reasoning level. But, due to the general design of Larch and its first-order logic foundation, this is just an abbreviation mechanism. The notations must be dissolved in a preprocessing step that transforms Larch specifications into first-order formulas for the proof in the LSL prover LP. That is, structures do not exist at the level of the prover and its language, instead they are all dissolved into simpler constructs of a first-order logic. We lose the advantage of the notation for the actual proving process.

Generally, PVS and IMPS are restricted to a fixed logic: classical and three-valued [Far90] higher order logic, respectively. In PVS the theory level is separate from the reasoning level, which causes severe restrictions in the adequate treatment of abstract algebra, as we will see in the following section. Compared to IMPS, its theory concept is inferior. The strict hierarchies forbid self-reference, which is possible in IMPS (*cf.* Section 2.2.1). It is not possible to do such advanced transformations between sorts and terms as in IMPS (*cf.* Section 2.3.4). IMPS is more flexible but its higher order logic with undefinedness may be rather inconvenient in some cases.

IMPS has definitely the most adequate module system in the first group of provers we examined. The theory interpretation mechanism is a well developed and powerful tool. It enables a kind of flexible polymorphism. Abstraction can be modelled *via* theory interpretation (see Section 2.2.1), although not as explicitly stated as in a system that offers polymorphism. A sort **s** stands for a constant type. One can express properties quite neatly, because they can all refer to this **s**. And, by the possibility of theory in-

terpretation, this constant value can always be interpreted as any sort in another theory, linking all the results proved in the base theory and its extensions. It is a clever way of avoiding difficult checking of polymorphic variables by leaving that open for the time, when the abstraction is actually made use of, *i.e.* at the point when we want to interpret `s`.

Although IMPS has an elaborate theory concept, the general approach does not seem ideal to us. Much of the properties *about* structures that are expressed as theories has to be formulated *via* translation of theories. This does not seem to be very explicit. The obligations that arise from theory interpretation are the ones that are of quite central concern, especially in abstract algebra, where propositions about structures are ubiquitous. This major issue is hidden in the general mechanism of theory interpretation. In the application to abstract algebra in the subsequent section, we will see that this causes a certain clumsiness in modular reasoning.

2.3 Abstract Algebra with Modules

A basic example to illustrate modules is the algebraic theory of *groups*. It is used in all of the three provers compared in Section 2.2 to advertise the advantages of their theory concepts. To explain the convenient use of modules for abstract algebra, we consider the example of groups in the three provers more closely.

A group is a set G together with a binary relation $\circ$ on this set. There is a neutral element e in G and for each element of G there exists an inverse in G. These properties, together with associativity, form the axioms of a group. This abstract and concise description immediately gives rise to the idea to model groups as a single theory in a theorem prover with modules. We can use parameters to abstract from the carrier G, the operation $\circ$, the identity e, and the inverse.

```
theory group (G, ∘, e, inv)
  axioms
    neutral_element : ...
    ...
end
```

Some approaches to formalizing abstract algebra assume the existence of the unit and the inverse rather than model them by parameters (*e.g.* [Gun89]). However, the systems PVS, IMPS, and Larch use parameterization. To be able to compare our developments with these examples we will use the parameterized version as well in the remainder of this work.

2.3.1 Groups in IMPS

The def-form introducing the language for a theory of groups is:

```
(def-language groups-language
  (base-types gg)
  (constants
    (e "gg")
    (mul "[gg,gg,gg]")
    (inv "[gg,gg]")))
```

The language definition introduces the sorts and constants. The definition of the theory of groups integrates this language and formulates the axioms:

```
(def-theory groups
  (component-theories h-o-real-arithmetic)
  (axioms
    (left-op-id
      "forall(x:gg, e mul x = x)")
    (right-op-id
      "forall(x:gg, x mul e = x)")
    (left-op-inv
      "forall(x:gg, inv(x) mul x = e)")
    (right-op-inv
      "forall(x:gg, x mul inv(x) = e)")
    (op-associativity
      "forall(x,y,z:gg, (x mul y) mul z = x mul (y mul z))")))
```

This example is taken from the on-line documentation for IMPS [FGT98][4]. The example of groups will also be considered in the other systems to point out differences. Because IMPS offers the most advanced formalism, we go into some detail here to find out how far we can reason explicitly with the notion of groups. The specification given above enables reasoning in the algebraic structure of groups. Sometimes we might want to state that a structure is a group in the logic. Therefore, an additional definition of the group axioms in form of a quasi-constructor is necessary:

```
(def-quasi-constructor group
  "lambda (gg%: sets[gg], mul%: [gg,gg,gg], e%:gg, inv%: [gg,gg],
           forall(x,y:gg, x in gg% and y in gg%
                  implies mul%(x,y) in gg%) and e% in gg% and
           forall(x:gg, x in gg% implies inv%(x) in gg%) and
           forall(x:gg, x in gg% implies mul%(e%, x) = x) and
           forall(x:gg, x in gg% implies mul%(x, e%) = x) and
           forall(x:gg, x in gg% implies mul%(inv%(x), x) = e%) and
           forall(x:gg, x in gg% implies mul%(x, inv%(x)) = e%) and
           forall(x, y, z:gg,
```

[4]Note that the axiomatization of groups is not minimal: the "right" versions of the axioms can be derived.

```
                ((x in gg%) and (y in gg%) and (z in gg%)) implies
                 mul%(mul%(x, y), z) = mul%(x, mul%(y, z))))"
  (language groups-language))
```

This definition just introduces a λ-term which has no logical significance, *i.e.* it is not an axiom, just a definition. Since it is not a theory definition and does not define any base sorts or constants it may be used in any context where enough items of appropriate signature exist to apply the λ-form to them. Statements like `group(G,*,1,-)` are then of type `prop` and may be part of formulas. They raise all necessary conditions.

Note that this is a second formalization of groups, separate from the first one. There is no implicit connection between the two — nor any guarantee that they are consistent. It would be nice to check consistency of the double formalization by theory interpretation, but this is not possible: one of them is not a theory but a quasi-constructor.

One example illustrating how the gap between theories, their types and an adequate representation as a set for algebraic structures is bridged in IMPS is given by the example of the definition of subgroups. Subgroups are not defined by a theory, instead by a constant predicate.

```
(def-constant SUBGROUP
  "lambda (a: sets[gg],
     nonempty_indic_q(a) and
     forall (g, h:gg, (g in a and h in a) implies (g mul h in a)
     and
     forall (g:gg, (g in a) implies (inv(g) in a)))"
  (theory groups))

(def-constant GG%SUBGROUP
  "sort_to_indic (gg)"
  (theory groups))
```

The second constant `GG%SUBGROUP` uses the quasi-constructor `sort_to_indic` to transform the sort `gg` — representing the base set of a group in the theory for groups — into a set, or an indicator in IMPS parlance. Only through this transformation it is possible to apply the predicate subgroup to groups. This is not realized by theory interpretation, instead by the mysterious quasi-constructor `sort_to_indic` (see Section 2.3.4).

```
(def-theorem GG-IS-A-SUBGROUP
  "subgroup (gg%subgroup)"
  (theory groups)
  (usages transportable-macete)
  (proof
   ...
))
```

The decisive step here is not theory interpretation, but the transformation of the base type `gg` into a set by application of `sort_to_indic`.

2.3.2 Groups in PVS

The following theory definition taken from [OSRSC98] declares a theory of groups:

```
groups [G : TYPE,
        e : G,
        o : [G,G->G],
        inv : [G->G]] : THEORY
BEGIN
  ASSUMING

    a, b, c: VAR G
    associativity : ASSUMPTION a o (b o c) = (a o b) o c
    unit : ASSUMPTION e o a = a AND a o e = a
    inverse : ASSUMPTION inv(a) o a = e AND a o inv(a) = e

  ENDASSUMING
END groups
```

With the PVS formalization of groups as a module, we can use theory instantiation and import facilities. We can use this theory to represent the idea of groups, *e.g.* we can instantiate the module, say to integers[5] `IMPORTING Group(Z, +, 0, -)` and produce the corresponding obligations. But this is not explicitly qualifying the tuple `(Z, +, 0, -)` to be a group. We cannot use this statement in a formula, because it is no logical term, just a PVS command. Consequently we cannot talk about all groups, or define anything depending on the group property. For example, we cannot use this definition of groups to define subgroups or reason about them. Unlike IMPS, PVS does not offer any other more adequate formalization of groups that could be used to model substructures like subgroups.

2.3.3 Groups in Larch

The example of groups can be expressed in LSL[6]:

```
Group: trait
introduces
__ o __:  T, T → T
unit:  → T
__^-1:  T → T
asserts ∀ x, y, z :T
(x o y) o z == x o (y o z);
unit o x == x;
```

[5] Assuming that there is a formalization of integers where Z is the base type

[6] Note the integration of the LaTeXfeatures in the code declaration, *e.g.* in declaring the inverse element

```
(x ^-1) o x == unit;
implies Monoid, Inverse
```

The **implies section** is added here to illustrate that it is possible to state properties which should be shown. The cited traits `Monoid` and `Inverse` contain properties about $\circ$ and $\_^{-1}$ which have been stated as axioms in the corresponding IMPS examples (*cf.* Section 2.3.1). Here, we incorporate them by the reference in the **implies** section and point out that they are entailed in the other properties.

Note that an **assumes** could have been alternatively stated at the beginning, similar to PVS, to state constraints on parameters. In that case we would have to have the groups constituents as parameters. Although we can use traits as statements, like `Monoid` in the **implies** section, this would never be possible in a formula, because LSL is based on first order logic. Thus, such statements about structures cannot be expressed at all in the logic.

2.3.4 Analysis of Group Specification in IMPS, PVS, and Larch

The abstract character of groups is modelled in the three systems by using (generic) sorts or explicit parameters to model the contents of the group. This parameterization enables the instantiation of the group theory to actual groups. Reasoning about properties of group elements and the operation $\circ$ is possible inside such a theory. The derived results can be reused by an instantiation of the theory of groups. This is the way of formalization we find in the examples of the documentation of the theorem provers IMPS, PVS, and Larch. It is an elegant way to model mathematical structures and abstraction using the concepts of modules provided by these theorem provers.

However, an adequate way of reasoning is not possible in this setting. Naturally, the possibility to abstract from the given representation is required. For example, we must consider the class of all groups to enable reasoning about general properties which hold, say, only for finite groups. This class of all groups cannot be defined here because the theory level is separate from the reasoning level; modules are not first class citizens. Furthermore, it is convenient to consider related structures as for example semigroups or rings which have much in common with groups but are structurally different. We do not only want to relate the structures on the theory level; we might need to prove that, for example, a semigroup is a group if further properties hold. This is not possible with the theory mechanisms of PVS and Larch. The only system that finds a way around this problem is IMPS.

The IMPS solution

As illustrated in section 2.3, we can transform the base type of a group module into a set by using a quasi-constructor called `sort_to_indic`. Then we can apply predicates like `subgroup` to the image of the base type. The remarkable map `sort_to_indic` uses the basic sort `uu` that is the "universal" sort used for all quasi constructors, and the element `an%individual` that is the only element of the predefined sort `unit%sort` we encountered earlier.

```
(def-quasi-constructor SORT-TO-INDICATOR
  "lambda(e:uu, lambda(x:uu, an%individual))"
  (language indicators)
  (fixed-theories the-kernel-theory))
```

Drawing from a personal communication with F. Thayer, one of the authors of IMPS, we are able to explain how this quasi-constructor works. The term

```
sort_to_indic(gg)
```

in the definition of the constant `GG%SUBGROUP` in Section 2.3.1 is parsed as

```
(SORT-TO-INDICATOR (undefined gg))
```

Now, the λ-expression `lambda(e:uu, lambda(x:uu, an%individual))` applied to `(undefined gg)` reduces to `lambda(x:gg, an%individual))` thereby interpreting `uu` as the base type of the group `gg`. This function, read as an indicator, and thereby as a set, returns `an%individual` for any element of type `gg`, *i.e.* is defined for any element of `gg`. Hence, this indicator represents the set that contains all elements of type `gg`. This is a very noteworthy device to transform types into sets, that can only be achieved by using special parsing combined with the notion of undefinedness.

We see that IMPS has to walk out of its way to get a complete characterization of the algebraic structure of groups. The obscurity of this solution shows that it is necessary to find a better method for an adequate representation of abstract algebraic structures.

2.4 Type Theory: LEGO and Coq

The systems Coq and LEGO are based on type theory. They implement variants of the Calculus of Constructions [CH88]. They do not have module systems comparable to IMPS, PVS and Larch. Instead they have a special type abstraction that may be used to represent structures. This abstraction is given by the concept of dependent types, in particular Σ-types. In Chapter 5 we will introduce these types in more detail.

Coq and LEGO are very similar. In this section we only describe the major characteristics of the two systems in order to lead into a description of the aspects of type theory that are important from our point of view.

There are other systems that are more elaborate, *e.g.* Nuprl [C+86]. The latter has an interesting notion of theories [Hic97] using a special form of dependent types [Hic96]. Although we will use dependent types as well (see Chapter 5), a plain consideration of the main mechanism is sufficient for our work.

2.4.1 Coq

The type theory on which Coq is based is called the Calculus of Inductive Constructions (CIC) [CPM90]. This is an extension of the Calculus of Constructions by primitive inductive definitions.

Besides many other methods and techniques in Coq, it has a *section* device [Dow90]. This device is part of a *Mathematical Vernacular*[7]. In Chapter 4 we describe a concept of *locales* for Isabelle that has something in common with sections.

2.4.2 LEGO

LEGO [LP92] models various different type systems. Among them are the calculus of constructions, that is also the basis for Coq, and the Extended Calculus of Constructions [Luo90b], ECC for short. Another type theory modelled by LEGO is the Unified Theory of Types [Luo92], abbreviated UTT. This type theory is an extension of ECC, and is also the one used for the case study in [Bai98] mentioned earlier (see Section 1.3). As an extension of ECC, the unified theory of types is one of the most powerful type theories, especially with respect to expressivity. Therefore, we selected UTT as the basis for a short introduction to type theory.

2.4.3 Type Theory

In the world of type theory the types are part of the logic, or *vice versa*, the logic is implemented by types. By the Curry-Howard isomorphism [SH80], the types are interpreted as logical formulas; this isomorphism is sometimes called *propositions-as-types* [How80].

The calculus of constructions can be seen as a typed higher order λ-calculus. Every term has a type. We write

$$a : A$$

for "term a has type A". The types can be again considered as terms. Hence, this two-sided relationship builds up a type hierarchy.

[7] Dowek describes a mathematical vernacular as a "language in which mathematics can be written both usable for publishing and mathematically fully described." He refers to N.G. De Bruijn.

The unified theory of types extends the calculus of construction by a *hierarchy* of type universes with *subsumption* and an *impredicative* type of propositions, amongst other things. In the following we explain these notions.

The hierarchy of type universes is a sequence $Type_0$, $Type_1$, $Type_2$, ... This hierarchy has that usually

$$Type_i : Type_{i+1}$$

Subsumptive means that each universe includes its predecessor.

$$A : Type_i \Rightarrow A : Type_{i+1}$$

This hierarchy of progressively higher type universes is a way to enable types to be given even to terms that quantify over whole type universes. An example for such a term is the Π-type that is a generalization of the function type. The type constructor Π abstracts over an entire type. But as long as B can be typed by $Type_{i+1}$, the type universes enable to type a Π-abstraction as follows

$$\Pi\ x : A.\ B : Type_{i+1}$$

Having abstraction over types, such as Π, it is important to have universes, if one wants to avoid having *impredicativity*. An impredicative definition is one that presupposes the existence of the object that is being defined [Bai98] and refers to it in the definition. An example is the type of propositions in UTT. This type is impredicative because one can build types using abstraction over the whole type of proposition *Prop*. An example is the following type, expressing the tautology $P \Rightarrow P$:

$$\Pi\ P : Prop.\ P : Prop$$

The abstraction $\Pi\ P : Prop.\ P$ is itself contained in the type *Prop*, although it abstracts over the entire type *Prop*. If all types were impredicative, then the type hierarchies would collapse, because no type construction would ever "leave" the type, *i.e.* go up one level. To express it *vice versa*: only open hierarchies enable us to type abstractions, like the Π-type, over predicative types.

The interesting point from our perspective is that, by the concept of type hierarchies, types can be terms again. Thus, on a higher level, it is possible to use types again in formulas. Alternatively, one may use impredicative types. The Π-types and Σ-types are useful to express structure. The fact that they are still typeable, and thereby terms on a higher level, makes such formalizations adequate.

For example, groups can be represented in UTT [Bai98] as

$$\Sigma\ G : set.\ \Sigma\ e : G.\ \Sigma \circ : map_2\ G\ G\ G.\ \Sigma\ ^{-1} : map\ G\ G.\ \text{group\_axioms}$$

where group_axioms abbreviates the usual rules for groups, which use the parameters G, e, $\circ$ and $^{-1}$. Since this Σ-type can be considered as a term in a higher type universe, we can use it in other formulas. Hence, this formalization of groups is logically adequate.

A more detailed description of the Π-types and Σ-types and their application for structure representation will be given in Chapter 5 when we use them ourselves.

2.5 Outlook

In this chapter we have seen examples of modules for theorem provers. They illustrate various properties of which the most important for our purpose is *locality*. That is, modules encapsulate properties and definitions of mathematical structures like groups such that those can be used conveniently and close to mathematical notation in a local context.

We have also seen that the module concepts of theorem provers comparable to Isabelle, *i.e.* IMPS, PVS, and Larch, have some weaknesses once it comes to *adequacy*. Although in IMPS, adequacy is in principle possible, we find its realization not satisfactory.

Type theory systems, represented here by Coq and LEGO, offer the concept of dependent types for the representation of modular structures. If the type theory is sufficiently powerful, these types can be considered as first class citizens, whereby we achieve adequacy.

Summarizing, we have illustrated that locality is desirable but should respect adequacy. In the following chapter we are going to consider a major case study to prove that for abstract algebra we do really need these features. The case study will also lead to a refined characterization of the requirements for modular reasoning in Isabelle.

Chapter 3

Sylow's Theorem

The first theorem of Sylow is proved in Isabelle/HOL. We follow the proof by Wielandt, which is more general than the original and uses a non-trivial combinatorial identity. The mathematical proof is explained in some detail in Section 3.1, leading on to the mechanization of group theory and the necessary combinatorics in Isabelle. We present the mechanization of the proof thoroughly, giving reference to theorems contained in Appendix A. After a general analysis of the experiment, we draw from the experience of the case study some requirements for abstract algebraic reasoning. They give rise to some tentative ideas for the further development towards a concept for modular reasoning in Isabelle.

The theorem is most easily described as the converse of Lagrange's theorem. Lagrange says that the order of a subgroup divides the group's order. Unfortunately, the direct converse,

> *if a number divides the group's order then there is a subgroup with corresponding order,*

is not generally true. But the theorem of Sylow gives us at least,

> *if p is a prime and p^α divides the order of the group then there is a subgroup of order p^α.*

The proof of the theorem of Lagrange has been performed with the Boyer-Moore prover [Yu90]. Gunter formalized group theory in HOL [Gun89]. In the higher order logic theorem prover IMPS [FGT93], some portion of abstract algebra including Lagrange is proved. Mizar's [Try93] library of formalized mathematics contains probably more abstract algebra theorems than any other system. But to our knowledge none of the known systems has proved Sylow's theorem. We always considered it as a theorem which is hard to prove already in theory and this definitely makes it an interesting challenge for theorem provers.

Yet, the main motivation to prove this theorem is to explore the reasoning with abstract structures. As pointed out in Section 1.5, Sylow's theorem

is quite well suited for this purpose as it uses different views of structures and applies combinatorial reasoning to derive properties of those structures. We use this example of abstract algebra to find out about reasoning mechanisms in algebra and thereby to refine the requirements for modular concepts.

3.1 The First Sylow Theorem

Sylow's theorem gives criteria for the existence of subgroups of prime power order in finite groups.

THEOREM 1 *If G is a group, p a prime and p^α divides the order of G then G contains a subgroup of order p^α.*

In the following we write $a \mid b$ for *a divides b* and $o(G)$ for the order of G.

The proof displayed here and used as the basis for the formal proof is due to Wielandt [Wie59]. It generalizes the original form found by the Norwegian mathematician Sylow in 1872 [Syl72]. We give the proof following [Her64] but go much more into detail to prepare the description of the formalization.
Proof

The proof is presented in three major parts. In the second part the existence of a subgroup of G having p^α elements is shown by constructing a subgroup H and the final part proves by combinatorial arguments that H actually has p^α elements. In the construction of the subgroup H, we define the set $\mathcal{M}$ of all subsets of G having p^α elements. We have to consider first a combinatorial argument about $\mathcal{M}$, which is used in the final part of the proof.

3.1.1 Combinatorial Argument

Let $\mathcal{M}$ be $\{S \subseteq G \mid card(S) = p^\alpha\}$. If $p^\alpha \mid o(G)$ then there is $m > 0$ such that $o(G) = p^\alpha m$ because G is a group and hence nonempty. The cardinality of the set $\mathcal{M}$ is then $\binom{p^\alpha m}{p^\alpha}$ because this is the number of ways one can pick a set of p^α elements out of G. We define the number r as the *maximum* natural number such that $p^r \mid m$, that is $p^{r+1} \not| m$.

In the following we show that $p^r \mid m$ *iff* $p^r \mid \binom{p^\alpha m}{p^\alpha}$. The following argument yields this equivalence for an arbitrary natural number:

$$\begin{aligned}\binom{p^\alpha m}{p^\alpha} &= \frac{p^\alpha m(p^\alpha m-1)\ldots(p^\alpha m-(p^\alpha-1))}{p^\alpha(p^\alpha-1)\ldots 1} \\ &= m\frac{(p^\alpha m-1)\ldots(p^\alpha m-(p^\alpha-1))}{(p^\alpha-1)\ldots 1}\end{aligned}$$

The power of p dividing $p^\alpha m - k$ in the numerator is the same as the power of p dividing $p^\alpha - k$ in the denominator for all $k = 1, \ldots, p^\alpha - 1$. This observation holds in the one direction because if $p^s \mid p^\alpha - k$ and $k < p^\alpha$ then $s \leq \alpha$ and thus for the quotient x, *i.e.* $p^s x = p^\alpha - k$, we can construct $p^\alpha m - k$ as $p^s(x + (p^{\alpha-s}(m-1)))$ whereby $p^s \mid p^\alpha m - k$.

Conversely, a slightly more difficult argument yields that $s \leq \alpha$: if similarly $k < p^\alpha$ and $p^s \mid p^\alpha m - k$ assuming for contradiction that $p^s > p^\alpha$ then $p^\alpha \mid p^s$ and by transitivity of divisibility $p^\alpha \mid p^\alpha m - k$. Since $p^\alpha \mid p^\alpha m$ it must follow that $p^\alpha \mid k$ in contradiction to $k < p^\alpha$ (and $0 < k$). Thus, we can construct $p^\alpha - k$ from $p^s x = p^\alpha m - k$ (where x is again the quotient) as $p^s(x - (p^{\alpha-s}(m-1)))$. The property $s \leq \alpha$ is necessary because we are in $\mathbb{N}$ and thus $s + (\alpha - s) = \alpha$ *only if* $s \leq \alpha$.

Thereby, the powers of p in denominator and numerator all cancel out. Hence

$$p \not\mid \frac{(p^\alpha m - 1) \ldots (p^\alpha m - (p^\alpha - 1))}{(p^\alpha - 1) \ldots 1}$$

and thus, the power of p dividing $\binom{p^\alpha m}{p^\alpha}$ is the same as the power of p dividing m. The right hand side of the above formula is an integer because it equals $\binom{p^\alpha m - 1}{p^\alpha - 1}$.

3.1.2 Construction of the subgroup H

Consider the set $\mathcal{M}$ of all subsets with p^α elements in G. On this set, define a relation $\sim$ as $M_1 \sim M_2$ if there exists a $g \in G$ such that $M_1 = M_2 g$. It is straightforward to prove that this relation is an equivalence relation on $\mathcal{M}$. Now, for the maximum number r such that $p^r \mid m$ we claim that there is an equivalence class M in $\mathcal{M}/\sim$ such that $p^{r+1} \not\mid card(M)$. If not, then p^{r+1} would divide the cardinality of *all* classes in $\mathcal{M}/\sim$ and thus $p^{r+1} \mid card(\mathcal{M}) = \binom{p^\alpha m}{p^\alpha}$ because equivalence classes partition $\mathcal{M}$. But, this would yield by the combinatorial argument of Section 3.1.1 that $p^{r+1} \mid m$ in contradiction to the assumption that r is maximal.

Now, let n be the cardinality of this class M. Since M has n elements, let us name them $M_1, \ldots, M_n$. We pick M_1 out of the equivalence class M — which is possible since $n \neq 0$ — and construct the subgroup H from this p^α-set M_1 as

$$H \equiv \{g \in G \mid M_1 g = M_1\}$$

This set H is a subgroup:

- $e \in H$ because $M_1 e = M_1$ for all subsets of G and thus also for M_1.
- for $a, b \in H$ is $M_1 a = M_1$ and $M_1 b = M_1$ by the definition of H. Thereby, $M_1(ab) = (M_1 a)b = M_1 b = M_1$ yields $ab \in H$.

These two criteria are sufficient to show that H is a subgroup.

3.1.3 Cardinality of H is p^α

First we establish that $n \cdot o(H) = o(G)$, in other words $card(M) \cdot o(H) = p^\alpha m$. To this end we construct a bijection between M and the set of right cosets G/H of H. By construction of H, we get the equivalence:

$$(Ha = Hb) \equiv (ab^{-1} \in H) \equiv (M_1 ab^{-1} = M_1) \equiv (M_1 a = M_1 b)$$

for all $a, b \in G$. That is, whenever a and b are in the same right coset of H (or their cosets are equal, respectively) they form the same $M_1 a = M_1 b$, name it N; and $N \in M$ because $Nb^{-1} = M_1$, hence $N \sim M_1$. So $Ha \mapsto M_1 a$, for all $a \in G$, defines a mapping from G/H to M. Since $N \in M$, N is some M_j, $j \in \{1, \ldots, n\}$, and conversely, each M_j is of the form $M_1 a$ for some $a \in G$ by definition. So the mapping $Ha \mapsto M_1 a$ for all $a \in G$ is in fact a bijection.

By this bijection we know that $card(M) \cdot card(H) = card(G/H) \cdot card(H)$ which equals $o(G)$ according to Lagrange's theorem (*cf.* Theorem 2).

Now we prove the two directions separately:

1. $p^\alpha \leq o(H)$:

   We constructed M such that $p^{r+1} \nmid n = card(M)$. Hence, for the maximum k such that $p^k \mid n$ must hold $k \leq r$, *i.e.* $p^k \mid p^r$. By construction of r holds $p^{\alpha+r} \mid p^\alpha m = n \cdot o(H)$ and consequently $p^{\alpha+k} \mid n \cdot o(H)$. But since k was already the maximum power of p dividing n we get from this $p^\alpha \mid o(H)$ whereby $p^\alpha \leq o(H)$.

2. $o(H) \leq p^\alpha$:

   For some arbitrary $m_1 \in M_1$, we have $m_1 h \in M_1$ for all $h \in H$ because of the definition of H as $\{g \in G \mid M_1 g = M_1\}$. Since this group operation is an injection, *i.e.* $h_1 \neq h_2 \Rightarrow m_1 h_1 \neq m_1 h_2$ (cancellation law for the binary operation), it follows that M_1 must have at least $o(H)$ different elements whereby $o(H) \leq card(M_1)$. Since the set M_1 is in $\mathcal{M}$ it has p^α elements and thereby finally $o(H) \leq p^\alpha$.

Summarizing, the constructed subgroup H has exactly p^α elements. □

The original form of the theorem of Sylow is a special case of the previous one:

COROLLARY 1 *If G is a group, p a prime, $p^m \mid o(G)$ and $p^{m+1} \nmid o(G)$ then G contains a subgroup of order p^m.*

As we have seen in the previous proof, the property of the m here being the maximal power of p dividing the order of G had been internalized, which made the theorem more general but the proof harder (in the combinatorial argument).

3.2 Formalization of Groups in Isabelle/HOL

The proof of Sylow's theorem demands a formalization of groups on a fine scale. We need to consider the carrier of the group as a set which has different kinds of subsets, *e.g.* subgroups, cosets, and arbitrary subsets of certain cardinalities. These subsets play a rôle in the reasoning about the group factorization in terms of the equivalence relation, which is used in the construction of the Sylow subgroup. Hence, we have to be able to view the constituents of the group from some completely different perspectives.

In the following, we explain the formalization of groups we used to handle these different views on groups in our experiment.

3.2.1 Groups in Isabelle

For the formalization of Sylow's theorem, we used the theory HOL of Isabelle. We preferred it to pure set theory represented by ZF because we wanted to employ polymorphism for the abstraction over the base set of a group. HOL offers a formulation of typed sets. Sets are here basically a syntactical abbreviation for predicates. By switching between the set representation and the corresponding predicate, we can combine the convenience of mathematical notation with the power of higher order logic reasoning with types.

To encapsulate the definition of a group by its operations and corresponding axioms, we employ the following definition of groups as a typed set of quadruples:

```
Group_def "Group ==
{(G,f,inv,e). f ∈ G -> G -> G & inv ∈ G -> G & e ∈ G &
          (∀ x ∈ G. ∀ y ∈ G. ∀ z ∈ G. (f (inv x) x = e) &
               (f e x = x) & (f (f x y) z = f (x) (f y z))))}"
```

The constant `Group` is a typed higher order logic set. Its type is

```
('a set * (['a, 'a] => 'a) * ('a => 'a) * 'a) set
```

The symbol `'a` symbolizes a type variable, *i.e.* the group definition may be instantiated to arbitrary types. The function constructor `A -> B` constructs the set of functions from a set `A` to `B`. We define this function set for HOL in more generality (including dependency) in Chapter 5.

3.2.2 Basic Properties

The definition of `Group` admits stating that a term `G` is a group quite concisely as `G ∈ Group`. Unfolding the definition of `Group`, represented by the above set, yields all the defining properties for the constituents of `G`. Naturally, we need to define projection functions for these constituents. Generally, they are available by the projection functions for the product type

of HOL, *e.g.* the function `fst` would return the set underlying the group. But, to make notations more self-explanatory we overload the projections for the quadruple `(G,f,inv,e)` with names symbolizing the meaning of the constituents: `carrier`, `bin_op`, `inverse`, and `unity`. To enhance the readability of formulas we define the pretty printing syntax `_.<cr>`, `_.<f>`, `_.<inv>`, and `_.<e>` for these projections. It can be applied as postfix in record-like fashion.

We have to derive the axioms from the definition — an additional cost for the neat representation — but this is very schematic and could be optimized by Isabelle's tactics to a high extent.

The definition of groups only assumes the minimal axioms, *e.g.* for the inverse only the *left* inverse rule $a^{-1}a = e$. We derive from the group definition a number of corresponding meta-level rules. For example,

```
[| G ∈ Group; a ∈ (G.<cr>) |] ==> (G.<f>)((G.<inv>) a) a = (G.<e>)
```

is the left inverse rule. For the closure properties, *e.g.* $inv \in G \to G$, we derive a more applicable rule from the definition, for example `inverse_closed`:

```
[| G ∈ Group; a ∈ (G.<cr>) |] ==> (G.<inv>) a ∈ (G.<cr>)
```

The symmetric properties, like the right inverse rule are then derived from them in the classical way: first we prove the left cancellation law for the binary operation $xy = xz \Rightarrow y = z$ and from that the symmetric unity rule $ae = a$; now, we can prove that $aa = a \Rightarrow a = e$ and by that the symmetric inverse rule. Finally, we can prove with the latter two the right cancellation law.

3.2.3 Subgroups

Building onto the basic properties of groups we consider the notion of a subgroup using the syntax `H <<= G` for *H is subgroup of G*. In the definition of the subgroup property we can use an elegant approach which reads informally: *a subset H of G is a subgroup if it is a group with G's operations.*

```
subgroup_def "H <<= G ==
  H <= (G.<cr>) & (H, (G.<f>), (G.<inv>), (G.<e>)) ∈ Group"
```

It is not completely trivial that this definition is possible because it depends on the way that groups are formalized: we can use the structure `Group` in a formula because it is a term.

Basic derived results are `SG_unity` — the unit of G is an element of every subgroup — from which we get that a subgroup is nonempty, or `0 < card(H)`. Related theorems about subgroups are that they are closed under product, *i.e.* $a, b \in H \Rightarrow ab \in H$ and under inverse, *i.e.* $a \in H \Rightarrow a^{-1} \in H$.

An introduction rule for the subgroup property is `subgroupI`:

```
[| G ∈ Group; H <= (G.<cr>); H ≠ {}; ∀ a ∈ H. (G.<inv>) a ∈ H;
   ∀ a ∈ H. ∀ b ∈ H. (G.<f>) a b ∈ H|] ==> H <<= G"
```

That is, for a nonempty subset H of a group G it is sufficient to check that it is closed under inverse and the binary operation to conclude that H is a subgroup. Ideally, we want to have an introduction rule where it is sufficient to show that a nonempty subset of G is closed under `G.<f>` to gain the subgroup property. Actually, this is the characterization of subgroup used in the Sylow proof (see Section 3.1.2). But, this result uses an argument about finite sets and repetitions of a^n for $n \to \infty$ if G is finite, which is quite complicated to prove formally. On the other hand, it is straightforward to prove the additional closure under inverse construction for the Sylow subgroup. Hence, we deviate at this single point from the mathematical proof of Sylow's theorem by using the longer characterization `subgroupI` (*cf.* Section 3.3).

The proof of Sylow's theorem uses Lagrange's theorem as well as an equivalence relation which is ranging over subsets with p^α elements. For both we need the notion of *cosets*.

3.2.4 Factorization of Groups

If H is a subgroup of a group G then the *right coset* of a with respect to H in G, written Ha, is the set $\{ha \mid h \in H\}$. We consider only right cosets here and sometimes refer to them as just *cosets*. The division of a group into cosets is a partition. The coset construction is needed when we consider so-called factorizations of a group. Then we look at H, the factor, as the unit and each coset as a member of the factorization with respect to the induced operation on cosets. An interesting point is to find out how the induced operation behaves on the factorization. For example, one can reason about the criteria which make the factorization G/H together with the induced operation on the cosets again a group (see Section 6.3.1).

Although the construction of a group factorization is defined merely for subgroups, it can as well be applied to arbitrary subsets of groups. Hence, in our definition we leave out the condition that the factor is a subgroup and define `r_coset` as

```
r_coset G H a == {b. ∃ h ∈ H. (G.<f>) h a = b}
```

The definition is equivalent to $\{ha \mid h \in H\}$.

To be able to talk about the factorization of a group into cosets, we further define the set of right cosets G/H as:

```
set_r_cos G H == {C. ∃ a ∈ (G.<cr>). C = r_coset G H a}
```

The notation `r_coset G H a` is not very satisfying. It is necessary to quote the group `G` for which we consider the coset construction. The mathematical

notation is just Ha, where the group G to which we refer should be clear from the context. Here, we need more notational support to get at least some notation like `H #> a`. This issue is addressed when we perform the actual Sylow proof (see Section 3.3). For the purpose of general group results, we deal with the verbose notation.

To prepare for the reasoning with cosets we derive some theorems about cosets (see Appendix A). They are partly concerned with the arithmetic for the induced operation: `coset_mul_assoc`, `coset_mul_unity`, `coset_join1`, `coset_join2`, `coset_mul_invers1`, and `coset_mul_invers2`. Further results are: the union of the set of all cosets equals the group (`set_r_cos_part_G`), cosets are subsets of G (`r_cosetGHa_subset_G`), cosets have equal cardinality (`card_cosets_equal`), unequal cosets are disjoint (`r_coset_disjunct`), and the set of cosets is a subset of the powerset of G (`set_r_cos_subset_PowG`).

The last few of these general results join together to prove Lagrange's theorem as we shall see in the following section.

3.2.5 Lagrange's Theorem

In contrast to the formalization of Yu [Yu90], the form of Lagrange that we need here is not just the one stating that the order of the subgroup divides the order of the group but instead gives the precise representation of the group's order as the product of order of the subgroup and the *index* of this subgroup in G, *i.e.*

THEOREM 2 *If G is a finite group and H is a subgroup of G, then $o(G) = |H| * |G/H|$*

The term G/H stands for G *modulo* H and is the factorization of G in right cosets of H. Its cardinality $|G/H|$ is defined as *index of H in G*. We sketch the proof here.

Proof

The proof of this theorem in our Isabelle formalization of group is quite straightforward. The basic idea is to reduce it to theorems about cosets using a fact that we can derive in general for finite sets (`card_partition`):

```
[| finite C; finite (Union C); ∀ c ∈ C. card c = k & finite c;
  ∀ c1 ∈ C. ∀ c2 ∈ C. c1 ≠ c2 --> c1 ∩ c2 = {} |]
  ==> k * card(C) = card (Union C)
```

Application of this to the original conjecture leaves us with the following subgoals:

```
1. finite (set_r_cos G H)
2. finite (Union set_r_cos G H)
3. ∀ c ∈ set_r_cos G H. card c = k & finite c;
4. ∀ c1 ∈ set_r_cos G H.
   ∀ c2 ∈ set_r_cos G H . c1 ≠ c2 --> c1 ∩ c2 = {}
```

The group G is finite by assumption. The subgoals 3 and 4 are the rules mentioned in the previous section: the cardinality of cosets is equal, and since they are subsets of G they are all finite. Their intersection is pairwise empty. Another derived result states that the powerset of a finite set is finite. Together with `set_r_cos_part_G` and `set_r_cos_subset_PowG` we get also 1 and 2. The finer scale of formalization with groups as first class citizens enables us to derive Lagrange's theorem in a general form and the proof is still concise. □

3.3 Sylow's Theorem in Isabelle/HOL

As mentioned in Section 3.2.4, the syntax for cosets is not very intelligible. Also, we would like to have a nicer notation for the group's binary operation. So far we must write `(G.<f>)a b` for ab. Since we need to quote the group G, the only way around this difficulty at the present state of Isabelle seems to abuse the theory file technique and declare a constant `G` and a type `i` of elements of `G` to build a context for the Sylow proof. To get nicer syntax and also to avoid repeating global premises in each lemma of the proof of Sylow's theorem — *i.e.* to save listing:

```
G ∈ Group, finite (G.<cr>), p ∈ prime, o(G) = (p ^ a)* m
```

— we define a theory `Sylow.thy` which contains a type `i`, a constant `G`, and the above premises as rules. Generally, the proof can be easily transformed into the explicit version abandoning the syntactical improvements, hence our approach here does not influence the soundness of the formalization of Sylow's theorem. Nevertheless, it disables a general application of the theorems derived in the theory `Sylow.thy` because they are not polymorphic (*cf.* Section 3.4.2). Chapter 4 will present *locales*, an extension to Isabelle, that builds temporary contexts, giving us the same notational benefits without sacrificing generality.

With this context at hand, we can abbreviate the verbose notation for `bin_op` and `r_coset` in terms of the constant `G` by the definitions

```
H #> x == r_coset G H x
x # y  == (G.<f>) x y
```

Furthermore, we can define `(G.<e>)` as `e` and `(G.<inv>) a` as `inv a`. In the Sylow theory we additionally define an identifier for the set $\mathcal{M}$ of p^α-subsets of G and for the equivalence relation $\sim$ over this set:

```
calM == {s. s ⊆ (G.<cr>) & card(s) = (p ^ a)}
RelM == {(N1,N2).(N1,N2) ∈ calM × calM &
                    (∃ g ∈ (G.<cr>). N1 = (N2 #> g))}
```

3.3.1 Prerequisites

Besides the theorems about groups and cosets, we need to derive properties about finite sets and cardinalities for this case study. Furthermore, some additional arithmetical results are needed and functions of combinatorics, *e.g.* binomial coefficients, and divisibility rules, have to be defined or derived. In the present section we show some of these; when displaying the rules we often leave out preconditions to enhance readability. Sometimes introduced in the text, names of theorems may also be displayed in square brackets at the right margin. The function `primrec` defines primitive recursive functions.

Arithmetic

The arithmetical rules which have to be derived in addition to the already existing ones of the Isabelle theory `Arith` are mostly obvious but some are nevertheless nontrivial. For example, the rule

```
a - (b - c) = a - b + c
```

needs the additional assumptions `c` $\leq$ `b` and `b` $\leq$ `a` because `-` is the difference for natural numbers. Similarly, the left cancellation law for natural number multiplication

```
k * a = k * b ==> a = b
```

is not generally valid (`k` must not equal zero).

Apart from such additional theorems about already existing functions, we define for the present case study an integer power operation by

```
primrec
  m ^ 0 = m
  m ^ (Suc n) = m * m ^ n
```

The use of the primitive recursion allows us to derive typical properties of this power function.

Finally, we have to define divisibility and derive basic facts about it. The most advanced rule about divisibility we derive is `div_combine` — the main argument for the proof part of Section 3.1.3.1 is to show that $p^{\alpha} \mid o(H)$:

```
[|...; ¬(p ^ (r+1) | n); p ^ (a+r) | n * k |] ==> p ^ a | k
```

Prime numbers are defined as a set, letting us quantify over all primes. Observe the syntactical overloading of the operator `|`, once as divisibility and once as logical or:

```
prime == {p. 1 < p & (∀ a b. p | a * b --> (p | a) | (p | b))}
```

We do not need many extra theorems for primes.

Finite Sets and Cardinalities

We have to prove that the cardinality of a finite set A is less or equal the cardinality of a finite set B if there exists an injection from A into B. From this, we can immediately show that if there is a bijection from A to B then their cardinalities are equal (`card_bij`).

We derive the already mentioned `finite_Pow` — the powerset of a finite set is finite — and the counting theorem `card_partition` (see Section 3.2.5). Furthermore, we need in the Sylow proof that if a number k divides the cardinalities of all classes of a set S factorized by an equivalence relation then k divides the cardinality of S (`equiv_partition`).

Binomial Coefficients and k-subsets

The definition of the choose operator is inspired by the HOL tutorial [Hol] by taking the Pascal triangle for the definition of $\binom{n}{k}$ instead of a ratio of factorials. We define $\binom{n}{k}$ as

```
primrec
  (0 choose k) = (if k = 0 then 1 else 0)
  (Suc n choose k) =
    (if k = 0 then 1 else (n choose (k - 1)) + (n choose k))
```

Using this primitive recursive definition we can derive immediately

```
Suc n choose Suc k =                                    [chooseD_add]
(n choose Suc k) + (n choose k)
```

From that we get all other necessary prerequisites quite easily: `n_choose_0`, `zero_le_choose`, `less_choose`, `n_choose_n`, `choose_Suc`, and `n_choose_1` for the basic ones and a multiplicative decomposition:

```
k ≤ n ==>                                               [chooseD_mult]
Suc n * (n choose k) = (Suc n choose Suc k) * Suc k
```

From that we can derive the theorems

```
k ≤ n ==> (Suc n * (n choose k)) mod Suc k = 0
k ≤ n ==>                                               [choose_defT]
(Suc n choose Suc k) = (Suc n *(n choose k))div Suc k
```

which are decisive in the first combinatorial part of Sylow's proof.

We can now prove

```
card {s. s ⊆ M & card s = k} = (n choose k)              [n_subsets]
```

if `card M = n` and `k ≤ n`. In the induction scheme for finite sets, applied to `M`, we can use some `x` not in `M`. Using the decomposition

```
{s. s ⊆ insert x M & card s = Suc k} =
{s. s ⊆ M & card s = Suc k}
∪ {s. ∃ s1 ∈ {s. s ⊆ M & card s = k}. s = insert x s1 }
```

for x not in M, we can use the induction hypothesis to show that

```
card {s. s ⊆ M & card s = Suc k} = ((card M) choose (Suc k))
```

on the one hand. On the other hand, we construct a bijection between

```
{s. ∃ s1 ∈ {s. s ⊆ M & card s = k}. s = insert x s1}
```

and

```
{s. s ⊆ insert x M & card s = k}
```

provided that x is not in M. We use the induction hypothesis again to show that

```
card {s. ∃ s1 ∈ {s. s ⊆ M & card s = k}. s = insert x s1} =
((card M) choose k)
```

The cardinalities we derived for the two components of the decomposition match the formula **chooseD_add**. Hence, after showing that the cardinality of the union of two disjoint sets is just the sum of the cardinalities of these sets, we can apply **chooseD_add** to finish the proof of the theorem **n_subsets**.

Preparation for Combinatorial Argument

The combinatorial argument of Sylow's proof is formalized by first defining a maximum number predicate **max-n** as

```
max-n k. P(k) == @k. P(k)&(∀ m. k < m --> ¬ P(m))
```

Thereby, we can state the combinatorial argument as

```
(max-n r. (p ^ r | m)) =                                    [const_p_fac]
(max-n r. (p ^ r | (((p ^ a) * m) choose p ^ a)))
```

which is a natural way of encoding this proposition. Unfortunately, the **max-n** construct uses the Hilbert-operator **@** which names an element that fulfills a given predicate P but forces us to show the existence of such an element first. To make the proof easier we observe that the maximum power of p dividing a number n is the integer logarithm of n to the base p. By defining this integer logarithm function as

```
log_def "log p n == wfrec (trancl pred_nat)
              (λ f j. if ((0 < j) & (j mod p = 0))
                     then Suc(f (j div p)) else 0) n"
```

we gain the desired function that improves the derivation of the main proposition `const_p_fac`. The definition of this function uses the functional `wfrec` for well founded recursion. We first show some properties about this logarithm function to enable later calculations.

We show that this logarithm represents actually the maximum power of p dividing a number s by deriving

```
p ^ log p s | s                                          [max_p_div]
& (∀ m. log p s < m --> ¬(p ^ m | s))
```

This enables us to replace the `max-n` term by a `log` term and we can derive the unique existence of the logarithm (`unique_max_power_div_s`, `log_p_unique`, `max_p_div_eq_log`).

The theorem we finally use in the combinatorial argument is a combination of the latter ones, namely

```
[|...; ∀ r. ((p ^ r | a) = (p ^ r | b))|] ==> log p a = log p b
```

The results we need to calculate with the new logarithm operation are

```
n = (p ^ log p n)*(n div (p ^ log p n))
```

and

```
log p (a * b) = (log p a) + (log p b)                    [log_mult_add]
```

3.3.2 Proof

According to the structure of the mathematical proof, we present the formal proof of Sylow's theorem in three parts.

Combinatorial Argument

We have to show the conjecture `const_p_fac` (see Section 3.3.1). Using `unique_max_power_div_s` we can immediately reduce the conjecture to the logarithm equality:

```
log p m = log p ((p ^ a)* m choose p ^ a)
```

By `chooseD_mult` (or `choose_defT`, more precisely) this can be transformed into

```
log p m = log p ((p^a)* m *((p^a)*m-1 choose (p^a)-1) div (p^a))
```

Cancellation (`div_mult1`) yields:

```
log p m = log p (m * ((p ^ a)* m - 1 choose (p ^ a) - 1))
```

which can be decomposed by `log_mult_add` into

```
log p m = log p m + log p ((p ^ a)* m - 1 choose (p ^ a) - 1)
```

By arithmetical rules (add_0_right, add_left_cancel) we reduce the latter to

```
0 = log p ((p ^ a)* m - 1 choose (p ^ a) - 1)
```

which can be derived from:

```
¬( p | ((p ^ a)* m - 1 choose (p ^ a) - 1))
```

So far the calculation above is straightforward. That p does not divide the remaining ratio can also be shown following the outline of the mathematical proof. To this end, we derive the two directions. The forward direction is

```
[|...; k < (p^a); (p^r) | (p^a)* m - k |] ==> (p^r) | (p^a)- k
```

for which we have to derive `r` $\leq$ `a` as in the mathematical proof under the same premises as above. The backward direction,

```
[|...; k < (p^a); (p^r) | (p^a)- k |] ==> (p^r) | (p^a)* m - k
```

needs again `r` $\leq$ `a`. Now, we characterize ¬(`p | n choose k`) for the case `log p k = log p n` under more general preconditions (p_not_div_choose). Instantiating the latter to $p^\alpha m - 1$ and $p^\alpha - 1$ by plugging in the previous two lemmas we attain

```
¬(p | ((p ^ a) * m - 1 choose (p ^ a) - 1))      [const_p_fac_right]
```

which solves the combinatorial argument.

Construction of the subgroup H

By abusing the theory mechanism we gained the possibility to use more natural syntax for the algebraic operations. An additional cost for this nice representation is that we have to instantiate the rules derived for groups to the constants of the theory Sylow. The instantiations of the rules are marked by a leading I in their names, *e.g.* Ibin_op_closed is the instantiation of bin_op_closed to the proof context and reads

```
[| x ∈ (G.<cr>); y ∈ (G.<cr>) |] ==> x # y ∈ (G.<cr>)
```

Before we get into the concrete parts of the proof, we have to derive some facts about calM and RelM. First, RelM is an equivalence relation over calM. The proof is a straightforward check of the definition of equivalence relation.

The assumptions $M \in \mathcal{M}/\sim$, $p^{r+1} \nmid card(M)$, and $M_1 \in M$ are always made in the following derivations. They have to be proved finally, then they are discharged at the top level of the proof.

Under the assumption of M and M1 we prove now

```
{g. g ∈ (G.<cr>) & M1 #> g = M1} <<= G
```

Here we see that abusing the theory mechanism to get a nicer syntax pays off. The syntactical form of the constructed subgroup is very similar to the mathematical notation.

As already mentioned in Section 3.2.3 we use the alternative characterization `subgroupI` to tackle this task. It leaves us with some subgoals of which the first one is

```
{g. g ∈ (G.<cr>) & M1 #> g = M1} ≠ {}
```

It can be solved by showing that `e` is in this set. The two closure conditions, *i.e.* the potential subgroup is closed under the binary operation and under inverse construction, are derivable by insertion and definition expansion.

We avoid defining an abbreviation for the constructed subgroup in the proof because this construction is only visible inside the proof and vanishes at the top level.

Cardinality of H is p^α

The most difficult part of this subproof is to construct the bijection between M and `set_r_cos G H` (abbreviated G/H subsequently). Though mathematically it is sufficient to derive $(Ha = Hb) \equiv (M_1 a = M_1 b)$ and to define the actual bijection as the mapping $Ha \mapsto M_1 a$ for all $a \in G$, this is not as easy formally. The problem is, we have, for one direction, some M_j which has an equivalent form $M_1 a$ for some a. Unfortunately, we do not know this a but need to use it to construct the inverse image of $M_j = M_1 a$ as Ha. For the backward map we have the same problem.

We solve this problem by employing the Hilbert operator `@`, although this makes the proof quite messy. We reduce the bijection to two injections $f \in M \to G/H$ and $g \in G/H \to M$. For the sake of readability we use below H for {g. g ∈ (G.<cr>) & M1 #> g = M1}.

```
f ≡ λ M.  H #> (@g. g ∈ (G.<cr>) & M1 #> g = M)
g ≡ λ C. M1 #> (@g. g ∈ (G.<cr>) &  H #> g = C)
```

That is, we just define these maps by the properties we expect from them — this is like saying that the map is $Ha \mapsto M_1 a$ but is less clear; we do not know this a instead just describe it by its properties.

Though the terms in this proof grow quite large, making the derivation hard to read, the proof is again straightforward. Additional results needed in the course of the derivation are

```
card(M1) = card(M1 #> g)
(M1, M1 #> g) ∈ RelM
```

With the bijection at hand, we solve the index lemma:

```
card(M) * card{g. g ∈ (G.<cr>) & M1 #> g = M1} = card G [index_lem]
```

by reducing it to Lagrange's theorem. This reduction is performed by the derivation `card_M_eq_IndexH` which entails the constructed bijection `bij_M_GmodH`.

The two inequalities yielding $o(H) = p^\alpha$ can now be derived by plugging in all the prepared theorems according to the textbook proof.

1. `p ^ a ≤ card {g. g ∈ (G.<cr>) & M1 #> g = M1}`

   This task can be reduced by the divisibility lemma `div_order` to

   ```
   p ^ a | card {g. g ∈ (G.<cr>) & M1 #> g = M1}
   ```

   The main divisibility rule for this subproof (`div_combine`) leaves us with

   ```
   1. ¬(p ^ (max-n r. p ^ r | m)+ 1 | card(M))
   2. p ^ (a + max-n r. p ^ r | m) |
          card(M) * card {g. g ∈ (G.<cr>) & M1 #> g = M1}
   ```

   The first subgoal is entailed in the assumption about `M` for this proof. To the second one we can apply the previously derived index lemma to transform into

   ```
   p ^ (a + max-n r. p ^ r | m) | order(G)
   ```

   After replacing `order(G)` by `(p ^ a)* m` this can be reduced by basic arithmetic and divisibility rules to

   ```
   p ^ (max-n r. p ^ r | m) | m
   ```

   which is entailed in `max_p_div` (see Section. 3.3.1).

2. `card {g. g ∈ (G.<cr>) & M1 #> g = M1} ≤ p ^ a`

   We substitute `card(M1)` for `p ^ a`. By the preconstructed injection of M_1 into H (`M1_inj_H`), discussed in Section 3.1.3.2 the task is reduced to subgoals `finite M1` and `finite H` which can be solved by the already derived lemmas (see above).

The main proof finally puts together the parts `H_is_SG`, `lemma_leq1`, and `lemma_leq2` (the above inequalities 1 and 2).

Still, we have the two assumptions

```
M ∈ calM / RelM & ¬(p ^ ((max-n r. p ^ r | m)+ 1) | card(M))
M1 ∈ M
```

We discharge the former one finally in the top level proof `Sylow1` by applying `lemmaA1`. This lemma proves the existence of such an M by contraposition. Assuming for contradiction that such a set does *not* exist, we would have

```
∀ M ∈ calM / RelM. p ^ ((max-nr. p ^ r | m) + 1) | card M
```

But then, since equivalence classes are a partition (`equiv_partition`)

```
p ^ ((max-nr. p ^ r | m) + 1) | card calM
```

Since `card calM` $= \left(\begin{smallmatrix} p^\alpha m \\ p^\alpha \end{smallmatrix} \right)$ this contradicts `const_p_fac_right`.

The assumption `M1` $\in$ `M` is canceled by the theorem `existM1inM`. The existence of a set M_1 in the class M, whose cardinality is not divided by p^{r+1}, is derived from $M \in \mathcal{M}/\sim$. Since the empty set is *not* a member of $\mathcal{M}/\sim$, but M is a member, it follows that M is not the empty set; hence we can assume an $M_1 \in M$.

The formal proof of Sylow's theorem necessitates some smaller lemmas not visible in the textbook proof. They are mostly concerned with the existence of elements in the sets $\mathcal{M}$, M or M_1, inclusions between those sets and G and the cardinalities of those sets. Their names indicate this already: `zero_less_oG`, `zero_less_m`, `card_calM`, `exists_x_in_M1`, `M1_subset_G`, `finite_calM` (see Appendix A).

3.4 Conclusions and Requirements

As we have seen in the formalization of groups, there is a need to support structures to enhance the reasoning in abstract algebra. We modelled structures by defining a typed set for the structure of groups. This representation caused a little bit of extra work to retrieve the group properties from the definition (see Section 3.2.2). However, this representation is necessary to achieve logical adequacy, *i.e.* state and prove properties about groups. For reasons of notational adequacy we defined a separate theory for the definitions and assumptions of Sylow's theorem. This method is an abuse of the theory facility because we define a *constant* `G` to be able to define readable syntax. Furthermore, we abuse the possibility to state axioms in order to use them as local assumptions for the proof.

We begin in Section 3.4.1 with the summary of experiences from this case study and some conclusions on the performance and the statistics. Building on the observations about module concepts for theorem provers, we use the experience of the mechanization of Sylow's proof to propose a concept for the further direction of this thesis in Section 3.4.2.

3.4.1 Statistics and Experience

Sylow's theorem is a fundamental result of finite group theory. It usually stands at the end of a lecture course on group theory because it unifies most of the basic results about finite groups. A lot of textbooks on abstract algebra skip the theorem, or at least its proof, because it is difficult. The

proof by Wielandt that is the raw model for our formalization is quite concise (less than one page). Herstein [Her64], being a bit more detailed, uses almost two pages. In our introduction we need three pages.

The formalization of Sylow's theorem is quite a big experiment. Altogether we proved 278 theorems, of which only 52 are in the theory `Sylow`. The theory `Group` contains 121 theorems of which only 34 are concerned with group properties; the other 87 are dealing with the self defined operators: natural number logarithm, power, choose operator, the function set constructor `->`, and bijections (which use `->`). Though these are mostly arithmetical results, one could not expect them to be in the theorem library because the operators are rather exceptional and did not exist in Isabelle before. Another 104 theorems are extensions to the existing Isabelle theories of the HOL logic, sets, arithmetic, finite sets, equivalence relations, products and natural numbers. The total running time, if the entire development of the interactive proof is loaded into Isabelle, is five minutes on a 300MHz Penthium Pro.

The reasoning processes are quite subtle, mainly in the combinatorial argument. A lot of the reasoning deals with divisibility and finite set properties. As other formalizations show [PG96], reasoning with finite sets is tricky — though most arguments seem intuitively obvious. It necessitates a quite well structured analysis to avoid losing track of the proof. Often during the development we had severe problems because of innocuous side conditions. Some simple looking subtheorems turned out to be quite hard to prove. One could say that this is typical in proof development, but we think that it is obvious from the present chapter that the proof is just very subtle, though elegant.

Elegant constructions and proofs can be especially hard to mechanize. That is, they connect different domains of reasoning by unusual associations. This is intuitively appealing, but tedious for mechanical proofs. We are lucky if there is another unelegant but straightforward proof if we want to mechanize it.

What do we learn from the experiment? First of all, we think that hard proofs are good benchmarks for systems because they are well suited to reveal deficiencies of formal approaches. Tools or frameworks constructed for formal proofs might turn out to be not suited for intuitive associations connecting domains of mathematical reasoning. Especially, methods which prescribe the way of approaching a problem are naturally inflexible and can only be changed with unnatural complications to solve nonstandard tasks. Examples, for this are the module systems.

Another point which becomes obvious by performing big or difficult case studies is how far theory libraries are sufficiently equipped with theorems for such tasks. Surely, it is an impossible task to provide all possible lemmas which might arise in some obscure proof, but at least a certain level of knowledge about predefined theories must be provided. The Isabelle the-

ory library is mostly sufficient, though some probably not too frequently demanded theorems about equivalence relations and finite sets caused some additional difficulties.

Last but not least, we have to admit our admiration for mathematicians. While reconstructing a hard mathematical proof from the pure logic, one comes to points where it seems that there must be mistakes in the proof — but then it turns out that it is sound and just a bit trickier than expected. Relying on machine support to prove difficult theorems makes it almost impossible to believe that this can be performed soundly by just relying on a human brain which is often biased by the desire to solve the problem.

3.4.2 Sections

As we have seen in Chapter 2 and in particular in the present case study, modules enable local proof contexts by encapsulation, *i.e.* they provide locality[1]. But, we discover that the locality can be too permanent — rules of Isabelle are axioms, and definitions are meant to be of global significance. That is, theories are a too strong means to encapsulate local assumptions, like `G ∈ Group`. On the other hand algebraic structures like groups have to be presented as first class citizens anyway; modules as extra-logical devices are not appropriate here. We use typed sets which are sufficiently flexible to be adequate for groups. In the following we use the case study of Sylow's theorem to formulate a concept of sections. Those will enable encapsulation but more appropriately for the needs of algebraic reasoning.

The concept of a section described here is a tentative list of requirements; the actual construction will be considered in the following chapter. The concept resembles the ones in Coq [Dow90] and AUTOMATH [dB91, dB80].

Similar to the theory we defined for the main proof of Sylow's theorem, a section delimits a scope in which assumptions and definitions can be made and theorems depending on these assumptions are proved. They enable one to fix variables, like the group G, for the context of a proof, with special syntax if desired. Sections resemble modules but they are restricted to encapsulating terms of the logic, not types. They are of only local significance, *i.e.* they can be regarded as abbreviations for formulas of the meta-logic. A section entailing the assumptions $\Gamma_1, \ldots, \Gamma_n$ proving the theorems $t_1, \ldots, t_m$ may be regarded as the Isabelle meta-level formula

$$[\!|\ \Gamma_1; \ldots; \Gamma_n\ |\!] \Longrightarrow t_1 \& \ldots \& t_m$$

This enables an inside view of objects and theorems depending on section assumptions and definitions when the section is invoked; at the same time results shall have an outside representation, *i.e.* they can be globally applied.

[1] Although Isabelle theories are not very advanced modules they have to be considered as such in this context.

To fix and bind polymorphic constructors and definitions of groups, we need to declare a fixed type `i` in the Sylow case study (*cf.* Section 3.3). Isabelle's polymorphism is too general to fix a polymorphic constant for a group; the formalization would be unsound. Unfortunately, the resulting theorems are not polymorphic and hence not generally applicable to arbitrary groups. Another requirement for a section concept is thus to adapt Isabelle's polymorphism to the local context of a section.

The present case study can be seen as a prototype example for the use of sections. The theory we defined for Sylow's theorem shall be representable by a section. The constant G and the assumptions of the theorem are the contents. The outside view is the theorem in the usual form: all assumptions as premises of the meta-level. To realize the concept of sections the possibility of syntax definition in Isabelle has to be extended to allow us the same readable formalization as in the present case study. That is, we want to be able to use local definitions that depend on locally fixed values and we want to define pretty printing syntax involving local constants.

Summarizing our requirements we need a section concept that

- allows the assumption of local rules,
- enables local definitions that may depend on arbitrary but fixed values (like a group G) and possibly admit pretty printing syntax,
- creates a scope in which theorems depending on local assumptions can be proved,
- enables a reflection of the scope into a meta-logical formula, *i.e.* realizes an outside view of the section,
- allows to fix polymorphic constructors temporarily such that properties can be assumed soundly.

In the following Chapter we present the concept of *locales* that addresses this list of requirements.

Chapter 4

Locales

Drawing from the case study in the previous chapter we present a concept enabling locality. The concept is called *locales* because it realizes local contexts for proofs. Locales are implemented and have been released with Isabelle version 98-1 [KW98]. The implementation has been performed in collaboration with Markus Wenzel (TU-Munich).

Locales can be seen as an approach towards sectioning for higher order logic theorem provers, thereby addressing the requirements we stated in Section 3.4.2. After a motivating introduction in Section 4.1 we describe in Section 4.2 the locale concept and address issues of opening and closing of locales. We present aspects concerning concrete syntax, including a means for local definitions and continue in Section 4.3 with a detailed definition of locales and their features. Section 4.4 describes the implementation of the ideas. After summarizing the syntax in Section 4.5, we illustrate the application of locales in Section 4.6 by constructing a locale for groups. In Section 4.7, we consider some further steps we have explored towards the development of an explicit first class representation for locales. Finally, we discuss the further direction of the work in Section 4.8.

4.1 Motivation

In interactive theorem proving it is desirable to get as close as possible to the convenience of paper proof style. This makes developments more comprehensible and self declaring. In mathematical reasoning it is convenient to handle assumptions and definitions in a casual way. That is, a typical mathematical proof assumes propositions for one proof or a whole section of proofs and local to these assumption definitions are made that depend on those assumptions. The concept of *locales* is designed for the support of these processes of local assumptions and definition. Locales are a means to define local scopes for the interactive proving process. Fixed assumptions can be made that are visible inside the scope of the locale. When the locale

is invoked, theorems may be proved depending on these assumptions. A locale contains constants that enhance pretty printing syntax and enable local definitions in proof development contexts. Locales implement a sectioning device similar to that in AUTOMATH [dB80] or Coq [Dow90], but with optional pretty printing syntax and dependent local definitions added. Apart from the obvious difference that Coq and AUTOMATH are type theories, the concept of locales is static in that locales are declared in a theory and only opened interactively, whereas Coq and AUTOMATH sections are defined dynamically. Locales can be seen as a simple form of modules, yet they may be used for abstract reasoning and similar applications.

In mathematical proofs, we often want to define abbreviations for big expressions to enhance readability. These abbreviations might implicitly refer to terms, which are arbitrary but fixed values for the entire proof. Surely, Isabelle's pretty printing and definition possibilities are mostly sufficient for this purpose. But there are still examples where a definition in a theory is too strong: the syntactical constants used for abbreviations are of no global significance. Definitions in an Isabelle theory are visible everywhere.

In the case study of Sylow's theorem, we came across several such local definitions. For example, we defined a set $\mathcal{M}$ as $\{S \subseteq G_{cr} \mid card(S) = p^{\alpha}\}$ where G, p, and α are arbitrary but fixed values with certain properties. This is just for one big proof, and has no general meaning whatsoever. The formula does not even occur in the main proposition. Still, in Isabelle as it is, we only have the choice of spelling this term out wherever it occurs, or defining it on the global level, which is rather unnatural. On top of that, we need in a global definition to parameterize this example over all variables of the right hand side. In our example we would get something like $\mathcal{M}(G, p, \alpha)$ which is almost as bad as the original formula. This second drawback is because we cannot have something like *local constants*, here G, p, and α.

4.2 Locales — the Concept

Locales declare a context of fixed variables, local assumptions and local definitions. Inside the scope, theorems can be proved that may depend on the assumptions and definitions, and in which the fixed variables are treated like constants. The result will then depend on the assumptions of the locale, while locale definitions are eliminated.

A locale consists of a set of constants, rules and definitions. It is defined in an Isabelle theory and can be invoked in any session for that theory. An invocation assumes the locale rules; the locale definitions become visible. The rules can be used in the same way as usual Isabelle rules, *i.e.* like axioms. Similarly the definitions, like usual Isabelle definitions, abbreviate longer terms. But, the rules and definitions are local to the body of the invocation.

Theorems proved in the scope of a locale can be exported to the surrounding theory context. In that case, the rules used in its proof become assumptions of the exported theorem. The local definitions are expanded and then eliminated *via* generalization and reflexivity.

In the following, we explain several aspects of locales. There are basically two ideas that form the concept of locales: one is the possibility to state local assumptions, and the other one is to make local definitions which can depend on these assumptions and may use pretty printing. Those two main ideas depend on the notion of a locale constant.

4.2.1 Locale Rules

To explain what locales are, it is best to describe the main characteristics of Isabelle that lead to this concept and are the basis of the realization. Locale rules are based on Isabelle's concept of meta-assumptions.

In Isabelle each theorem may depend upon meta-assumptions. The theorem that ϕ follows from the meta-assumptions $\phi_1, \ldots, \phi_n$ is written as

$$\phi[\phi_1, \ldots, \phi_n]$$

The first main aspect of locales is to build up a local scope, in which a set of rules, the locale rules, are valid. The local rules are realized by using Isabelle's meta-assumptions as an assumption stack. Logically, a locale is a conjunction of meta-assumptions; the conjuncts are the locale rules. Opening the locale corresponds to assuming this conjunction.

In Isabelle-98, a meta-assumption can be used in proofs, but by the end of the proof, Isabelle would complain about the extraneous assumptions. With the locale concept added to Isabelle, locale rules become meta-assumptions when the locale is invoked. A theorem proved after a locale is opened can use these rules as axioms. They can be accessed by the names they have been given in the definition of the locale. At the end of a proof performed in a locale, the used rules become attached to the theorem as meta-assumptions. The theorem is admitted with the additional premises entailed implicitly. Hence, if this theorem is used in the same locale, the locale rules will not even be produced as subgoals. All locale rules can be used throughout the life cycle of the locale. The life cycle is determined by the interactive operations of opening and closing (see Section 4.3.2).

4.2.2 Locale Constants

A *locale constant* is an integral part of the locale concept. A locale implements the idea of "arbitrary but fixed" that is used in mathematical proofs. We can assume certain terms as fixed for a certain section of proofs and we can define other terms depending on them. These arbitrary but fixed terms are the locale constants. The locale constants may be viewed from

the outside as parameters because they become universally quantified variables when a major theorem is exported. Inside the locale they are just like normal constants and can thereby be used to enable pretty printing in local definitions. Locale definitions and rules may depend on these constants.

A locale corresponds to a certain extent to modules in a theorem prover, In contrast to the classical notion of a module, a locale does not contain type declarations and the constants are not persistent. The outside view of locales is realized in a different way. Instead of presenting the entire locale similar to a parameterized module that can be instantiated, one can export theorems of a locale. This export transforms a theorem into a general form whereby the locale is represented in the assumptions.

4.2.3 Locale Definition and Pretty Printing

A major reason for having locales is to make temporary abbreviations in the course of a proof development. As pointed out in Section 4.1, there are large formulas that are used in proofs and do not have a global significance. Conceptually, the definition of such logical terms is not persistent. Nevertheless, we want to use such definitions to make the theorems readable and the proof process clear. Hence, one aspect is the locality of these definitions. The other aspect, as illustrated by the introductory example as well, is that the local definitions might depend on terms that are constants in a certain scope. For example, we want to have $\mathcal{M}$ only, not a notation like $\mathcal{M}(G, p, \alpha)$, as would normally be necessary because we need to refer to the terms that form the other premises of the Sylow theorem.

Another common thing in abstract algebra are formulas which are not so big, but leave out information, *e.g.* we write Ha for the right coset of a with respect to the subset H of a group G. Since the group G containing this coset is a parameter to this definition we would have to define something like `r_coset G H a`. This is partly the same problem as with the parameters of the definition $\mathcal{M}$ but on top of it it requires pretty printing. The pretty printing facilities do not work here because we must not omit variables in definitions.

These features are realized by locales. In a locale where G is an arbitrary but fixed group, we can use the syntax `H #> a` instead of `r_coset G H a`. We create a definition mechanism for concrete syntax which implements the concept of a *local definition* for which we can define pretty printing syntax. The idea is to use the locale as a scope such that inside it a locale constant can be used to abbreviate longer terms. The terms we define can even be dependent on other locale constants if those are contained in the scope of the locale. Since locale constants are only temporarily fixed, the latter feature realizes dependent definitions, *i.e.* the defined terms may depend on implicit information of the context. This concrete syntax may only be used as long as the locale is open. Viewed from outside the locale, this syntax does not exist.

The theorems proved inside the locale using the syntax can be transformed into global theorems with the syntactical abbreviations unfolded and the locale constants replaced by free variables.

In a locale where we want to reason about a group `G` and its right cosets, we declare `G` as a locale constant. Then we can define an operation `#>` as another locale constant in terms of the underlying theory of groups where the operation `r_coset` is defined generally.

```
H #> x == r_coset G H x
```

When the locale containing this definition is open, we can use the convenient syntax `H #> x` for right cosets, and it refers to the fixed parameter `G`. If we prove a theorem in the locale scope and want to use it as a general result, we can export it. Then, the locale constant `G` will be turned into a variable, and the definition will be expanded to the underlying definition of right cosets.

4.3 Locales – Operations

4.3.1 Defining Locales

Locales are defined in an Isabelle theory. They can be opened in any descendant of that theory.

By combination of locale definitions, rules, and constants we attain a mechanism that constitutes a local theory mechanism. To adjust the dynamic idea of local definition and declaration to the declarative style of Isabelle's theory mechanism, we integrate the definition of locales into the theories. That is, a locale is another declaration part in Isabelle theory files. At first, we were toying with the idea of designing the entire concept in the interactive part of Isabelle. But, this would contradict the declarative nature of Isabelle's specification language, where every user defined entity resides in a theory. Consequently, locales are *defined* statically as an integral part of a theory. Yet, they are *used* interactively controlled by the locale operations (see 4.3.2).

To introduce the syntax, we show the example of groups. It uses the definition of groups as a typed set that we have encountered in the case study of Sylow (*cf.* Section 3.2.1). We use the type definition `'a grouptype` here. This is just an abbreviation for the type we used in the Sylow proof.

```
locale group =
  fixes
    G        :: "'a grouptype"
    e        :: "'a"
    binop    :: "'a => 'a => 'a"     (infixr "#" 80)
    inv      :: "'a => 'a"           ("i (_)"      [90]91)
  assumes
    Group_G   "G ∈ Group"
```

```
defines
  e_def     "e == (G.<e>)"
  binop_def "x # y == (G.<f>) x y"
  inv_def   "i x == (G.<inv>) x"
```

This part of an Isabelle theory introduces a locale for reasoning with an arbitrary but fixed group. The theory chunk introduced by the keyword `fixes` declares the locale constants in the familiar way, as in a `consts` declaration of Isabelle theory files. That is, besides their type, which is obligatory, there exists the option to define pretty printing syntax for the locale constants.

The subsequent `assumes` part specifies the locale rules. They are defined like Isabelle `rules`, *i.e.* by an identifier followed by the rule given as a string. Locale rules admit the statement of local assumptions about the locale constants. The `assumes` part is optional. Non-fixed variables in locale rules are automatically bound by the universal quantifier `!!` of the meta-logic. In the above example, we assume that the locale constant `G` is a member of the set `Group`, *i.e.* `G` is a group.

Finally, the `defines` part of the locale introduces the definitions that shall be available in this locale. Here, locale constants declared in the `fixes` section can be defined using the Isabelle meta-equality `==`. The definition can contain variables on the left hand side if the defined locale constant is a function. This improves natural style of definition, for example for constants that represent infix operators, *e.g.* `binop`. The non-fixed variables on the left hand side are considered as schematic variables and are generalized automatically. The right hand side of a definition must only contain variables that are already present on the left hand side. Thus far, locale definitions behave like theory level definitions. However, the locale concept realizes *dependent definitions* in that any variable that is fixed as a locale constant can occur on the right hand side of definitions. For example, a definition like

```
e_def "e == (G.<e>)"
```

contains the locale constant `G` on the right hand side. In principle, `G` is a free variable. Hence, this is a dependent definition. In Isabelle this would normally cause the error message "extra variable on right hand side". Naturally, definitions can already use the syntax of the locale constants in the `fixes` subsection. The `defines` part is, like the `assumes` part, optional.

Note also that there are two different ways a locale constant can be used; one is to state its properties abstractly using rules, and one is to declare it as a name for a definition. Actually, one locale constant can be used for both. In Section 6.4.2 we will see an example for this combined use.

4.3.2 Locale Scope

The definition of a locale is static, *i.e.* it resides in a theory. When the theory file is loaded the locales of this theory become analyzed and added to the internal representation of the theory.

Nevertheless, there is a dynamic aspect of locales corresponding to the interactive side of Isabelle. Locales are by default inactive. If the current theory context of an Isabelle session contains a theory that entails locales, they can be invoked. The list of currently active locales is called *scope*. The process of activating them is called *opening*; the reverse is *closing*.

Scope

The locale scope is part of each theory. It is a dynamic stack containing all active locales at a certain point in an interactive Isabelle session. The scope lives until all locales are explicitly closed. At one time there can be more than one locale open. The contents of these various active locales are all visible in the scope. In case of nested locales for example, the nesting is actually reflected to the scope, which contains the nested locales as layers. To check the state of the scope during a development the function `Print_scope` may be used. It displays the names of all open locales on the scope. The function `print_locales` applied to a theory displays all locales contained in that theory and in addition also the current scope.

The scope is manipulated by the commands for opening and closing of locales.

Opening

Locales can be *opened* at any point during an Isabelle session where we want to prove theorems concerning the locale. Opening a locale means making its contents visible by pushing it onto the scope of the current theory. Inside a scope of opened locales, theorems can use all definitions and rules contained in the locales on the scope. The rules and definitions may be accessed individually using the function *thm*. This function is applied to the names assigned to locale rules and definitions as strings. The opening command is called `Open_locale` and takes the name of the locale to be opened as its argument. In case of nested locales the opening command has to respect the nested structure (cf. Section 4.3.3).

Closing

Closing means to cancel the last opened locale, pushing it out of the scope. Theorems proved during the life cycle of this locale will be disabled, unless they have been explicitly exported, as described below. However, when the same locale is opened again these theorems may be used again as well,

provided that they were saved as theorems in the first place, using `qed` or ML assignment. The command `Close_locale` takes a locale name as a string and checks if this locale is actually the topmost locale on the scope. If this is the case, it removes this locale, otherwise it prints a warning message and does not change the scope.

Export of Theorems

Export of theorems transports theorems out of the scope of locales. Locale rules that have been used in the proof of an exported theorem become additional premises. The locale constants are universally quantified variables in these theorems, hence such theorems can be instantiated individually. Definitions become unfolded; locale constants that were merely used for definitions vanish. Logically, exporting corresponds to a combined application of introduction rules for implication and universal quantification. Exporting forms a kind of *normalization* of theorems in a locale scope.

According to the possibility of nested locales, there are two different forms of export. The first one is realized by the function `export` that exports theorems through all layers of opened locales of the scope. Hence, the application of export to a theorem yields a theorem of the global level, that is, the current theory context without any local assumptions or definitions.

The other export function `Export` transports theorems just one level up in the scope. When locales are nested we might want to export a theorem, but not to the global level of the current theory, instead just to the previous level because that is where we need it as a lemma. If we are in a nesting of locales of depth n, an application of `Export` will transform a theorem to one of level $n - 1$.

4.3.3 Other Aspects

Polymorphism

The polymorphism in Isabelle is such that definitions of constants with polymorphic types are individually quantified in front of the type expression. For example, a constant declaration

```
f :: 'a => 'a
```

means that the type of `f` is $\forall\alpha.\alpha \Rightarrow \alpha$. So, if there is a subsequent constant declaration using the same type variable α, those are different type variables. That is, they can be instantiated differently in the same context.

Now, for locales the scope of polymorphic variables is wider. The quantification of the type variables is placed outside the locale. So, type variables having the same name are actually the same variables. On the one hand, this difference allows us to define sharing of type domains of operators at an

abstract level. This is important for the algebraic reasoning that we are concentrating on. On the other hand, locale declarations are not polymorphic in the sense of Isabelle declarations.

This feature solves the problem we encountered in the Sylow case study and explained in Section 3.4.2. There we had to choose a fixed type `i` in order to model this restricted form of polymorphism.

Extension of Locales

A locale can be defined as the extension of a previously defined locale. This operation of extension is optional and is syntactically expressed as

```
locale foo = bar + ...
```

The locale `foo` builds on the constants and syntax of the locale `bar`. That is, all contents of the locale `bar` can be used in definitions and rules of the corresponding parts of the locale `foo`. Although locale `foo` assumes the `fixes` part of `bar`, it does not automatically include its rules and definitions. Normally, one expects to use locale `foo` only if locale `bar` is already active. The opening mechanism introduced in Section 4.3.2 is designed such that in the case of a locale built by extension it opens the ancestor automatically. If one opens a locale `foo` that is defined by extension from locale `bar`, the function `Open_locale` checks if locale `bar` is open. If so, then it just opens `foo`, if not, then it prints a message and opens `bar` before opening `foo`. Naturally, this carries on, if `bar` is again an extension.

An interesting device that has not yet been implemented is renaming for locale constants. This can be very useful if we want to have more than one instance of the same locale in the scope, for example when we reason with two different groups (*cf.* Section 6.3.2). The following illustrates a possible renaming mechanism: `loc_r` is created from `loc_c` by renaming all occurrences of locale constant `c` in `loc_c` by `r`.

```
locale loc_r = loc_c [r/c]
```

As of yet, it is an open question how to handle the pretty printing syntax in the case of renaming.

4.4 Implementation Issues

In this section we give a short description of the implementation of locales. We just describe some major features and how the concept is integrated with Isabelle and its theories. It uses a new method of generic theory data. Locales are another section of Isabelle theory files. The theory data functor is applied to a structure providing a package of necessary functions and types to build a new theory section. Then the functor transforms this basic package into one that adds to theories. The short description of the interface

for theory data is not meant to be complete, but is just necessary to illustrate the application of the method to the implementation of locales. The theory data functor has been created as a general tool by Wenzel. He collaborated with the author in the application of this method to provide an interface for locales. The implementation of locales based on this interface has been performed just by the author. Subsequently, we briefly describe the idea of locales as theory data, and then carry on to illustrate the implementation.

4.4.1 Theory Data

The functor `TheoryDataFun` in the file `Pure/theory_data.ML` provides a schematic way to produce new theory data. It is a generic method for adding new theory sections, *e.g.* inductive definitions, records, and now locales, to the theory interface of Isabelle.

The input signature of `TheoryDataFun` asks to provide a name and a type describing the new section. Furthermore, one has to plug in some standard operations, *e.g.* `empty` and `print`. From those the functor constructs an output structure that provides the type and the basic toolkit to implement the new theory section: a function `init` to initialize the section in theories, a new function `print` that can be applied to theories and print the new section, and two functions `get` and `get_sg` that return all section elements from a theory. In other words, provided some functions on the data type implementing a new section for Isabelle theory files, the functor integrates these functions into the theory data type and returns the interface to build up the new section as an extension to an Isabelle theory.

4.4.2 Interface and Locales

A locale definition forms a section of Isabelle theory files. Therefore, providing the input signature functions for the theory data functor, we can apply the functor and automatically get functions to initialize a theory with locales, print locales, and access them. This instantiation of `ThyDataFun` provides a starting point for the implementation of the concept.

The file `locale.ML` contains this creation of the interface. It also contains most of the other changes to Isabelle that implement locales. The function `add_locale` adding a locale to a theory, once it is read in by the appropriate theory parser routine, is the main function in this file.

The locale constants are realized by developing two layers of terms for them. Isabelle has two kinds of variables for terms: free variables, constructed by the datatype constructor `Free`, and schematic variables constructed by `Var`. The latter ones can be instantiated during unification, the former not. Basically the locale constants are `Frees` for which one can define mixfix syntax. We treat a locale constant `c` like `Free c` but produce for it a constant using the reserved name `\<^locale>c`. The pretty printing

syntax is assigned to the copy `\<^locale>c`. Proof terms in locales really contain the `Free c`, but for printing we use the constant `\<^locale>c`. For reading, we add translation functions permanently to the theory containing the locale.

The read and write functions of Isabelle have to be adjusted to locales: If a locale is open, we want any term that is read in to respect the bindings of types and terms of that locale. In `locale.ML` we augment the basic function `Thm.read_cterm` such that it checks if a locale is open, *i.e.* if the current scope is nonempty, and then bases the type inference on this information. Similarly, we adjust the function `pretty_term`, which is used to print proof states. Here, we replace now all `Free` constants `c` by their double `\<^locale>c`. Then the printing produces the pretty form.

4.4.3 Parsing

Another necessary part of the new method of adding a theory section to Isabelle is to provide a parsing method. The actual parser `locale_decl` for the locale definitions is just one ML-term constructed from parser combinators. It resides in the file `thy_parse.ML`.

Via `ThySyn.add_syntax` we can introduce locales to the general theory parser.

```
val _ = ThySyn.add_syntax
 ["fixes", "assumes", "defines"]
 [(section "locale" "|> Locale.add_locale" locale_decl)];
```

So, the keywords of a locale section are known, and for each section starting with the keyword `locale` the theory parser knows that the following has to be parsed with the parser `locale_decl` and the resulting raw form of the locale contents have to be added by `add_locale` to the theory.

4.5 Syntax and Functions

To provide an overview for subsequent examples, we summarize in this section the syntax and functions for locales described in this chapter. We use usual regular expression constructors $\{\}$, $|$, $[]$ and $*$. Terms enclosed in $\langle\rangle$ are non-terminals; typewriter font denotes terminals; quotation marks enclose single terminal symbols to avoid ambiguity. We use the lexical classes name, id, string, and mixfix of Isabelle [Pau94, Appendix A].

4.5.1 Syntax

Locale definitions can be added to Isabelle theories as

```
locale name   = [ name "+"]
 fixes        ⟨consts⟩
 [ assumes    ⟨rules⟩ ]
 [ defines    ⟨defs⟩ ]
```

The locale constants have the same format as Isabelle constant declarations.

$$\langle consts \rangle \equiv \{ \text{ name ``::'' string [ ``('' mixfix ``)'']}\}^*$$

Similarly, the locale rules can be defined like Isabelle rules.

$$\langle rules \rangle \equiv \{ \text{ id string }\}^*$$

Definitions have the same outer syntax as general rules.

$$\langle defs \rangle \equiv \{ \text{ id string }\}^*$$

4.5.2 Functions for Locales

Summarizing, the functions for the interactive use of locales are

```
Open_locale   : xstring -> unit
Close_locale  : xstring -> unit
export        :     thm -> thm
Export        :     thm -> thm
thm           : xstring -> thm
Print_scope   :    unit -> unit
print_locales:  theory -> unit
```

In particular,

- `Open_locale` *xstring*;

  opens the locale *xstring*, *i.e.* adds it to the scope of the theory of the current context. If the opened locale is built by extension, the ancestors are opened automatically.

- `Close_locale` *xstring*;

  eliminates the locale *xstring* from the scope if it is the topmost item on it, otherwise it does not change the scope and produces a warning.

- `export` *thm*;

  expands locale definitions in *thm*. Locale rules that were used in the proof of *thm* become part of its individual assumptions. This normalization happens with respect to *all open locales* on the scope.

- `Export` *thm*;

  works like `export` but normalizes theorems only up to the previous level of locales on the scope.

- `thm` *xstring*;

  applied to the name of a locale definition or rule it returns the definition as a theorem.

- `Print_scope ();`

  prints the names of the locales in the current scope of the current theory context.

- `print_locale` *theory*;

  prints all locales that are contained in *theory* directly or indirectly. It also displays the current scope similar to `Print_scope`.

4.6 Application Example from Abstract Algebra

We illustrate the use of locales by an example with the abstract algebraic structure of groups. Given that the Isabelle theory for groups contains the locale displayed in Section 4.3 we can now use it in an interactive Isabelle session. When the theory of groups is loaded, we can open the locale `group` with the command

```
Open_locale "group";
```

Now the assumptions and definitions are visible, *i.e.* we are in the scope of the locale `groups`. Using the print command, Isabelle displays the open locales of a theory.

```
print_locales Group.thy
```

It returns all the information about the locale `groups` and the scope.

```
locale name space:
  "Group.group" = "group", "Group.group"
locales:
  group =
    consts:
      G :: "'a set * (['a, 'a] => 'a) * ('a => 'a) * 'a"
      e :: "'a"
      binop :: "['a, 'a] => 'a"
      inv :: "'a => 'a"
    rules:
      Group_G: "G ∈ Group"
    defs:
      e_def: "e == (G.<e>)"
      binop_def: "!!x y. binop x y == (G.<f>) x y"
      inv_def: "!!x. inv x == (G.<inv>) x"
current scope: group
```

Note how the definitions with free variables have been bound by the universal quantifier of the meta level `!!`. The locale print function also gives information about the name spaces of the table of locales in the theory `Group` and displays the contents of the current scope.

As an illustration of the improvement we show how a proof for groups works now. We demonstrate a proof that shows how the inverse distributes over the group operation.

```
Goal "[|x ∈ (G.<cr>); y ∈ (G.<cr>)|] ==> i(x # y) = (i y)#(i x)";
```

Isabelle sets up the proof and keeps the display of the dependent locale syntax.

```
1.!!x y.[|x ∈ (G.<cr>); y ∈ (G.<cr>)|] ==> i(x # y) = (i y)#(i x)
```

We can now perform the proof as usual, but with the nice abbreviations and syntax. We can apply all results which have been previously derived in the same locale. In that case, the definitions can be used and locale assumptions will not appear as subgoals. We can even use the syntax when we use tactics that use explicit instantiation, *e.g.* `res_inst_tac`. When the proof is finished, we can assign it to a name using `result()`.

```
val inv_prod = "[| ?x ∈ (G.<cr>); ?y ∈ (G.<cr>) |]
   ==> inv (binop ?x ?y) = binop (inv ?y) (inv ?x)
   [!!x. inv x == (G.<inv>) x,
     G ∈ Group,
    !!x y. binop x y == (G.<f>) x y,
     e == (G.<e>)]" : thm
```

As meta-assumptions of the theorem we find all the used rules and definitions. The syntax uses the explicit names of the locale constants, not their pretty printing form.

If we want to export the theorem, we just type `export inv_prod`.

```
"[| ?G ∈ Group; ?x ∈ (?G.<cr>); ?y ∈ (?G.<cr>) |] ==>
 (?G.<inv>)((?G.<f>)?x ?y) = (?G.<f>)((?G.<inv>)?y)((?G.<inv>)?x)"
```

The locale constant `G` is now a free schematic variable of the theorem. Hence the theorem is universally applicable to all groups. The locale definitions have vanished. The other locale constants, *e.g.* `binop`, are replaced by their explicit versions, and have thus vanished together with the locale definitions.

Before we summarize the exposition of locales and discuss it in Section 4.8, we describe in the subsequent section a further development that we have experimented with but decided not to integrate into the locale concept.

4.7 Locales as First Class Citizens?

In this section, we introduce a line of research we followed towards the automatic generation of a first class representation for locales. What we want to gain is a reference to the locale in a logical sense. We explain the ideas, point out some advantages and explain why we finally abandoned this additional feature of the concept.

Although we used the set description for groups so far, we might think of employing a locale to represent groups in the style of modules as seen in Section 2.3.

```
locale group =
  fixes
    G :: "'a set"
    f :: "'a => 'a => 'a"
    ...
  assumes
    f ∈ G -> G -> G
    ...
    ∀ x ∈ G. (f e x = x)
```

We think of this structure as an abstract characterization of the property of the parameters `G`, `f`, `inv`, and `e`. Thus, we are not only interested in having an *ad hoc* mechanism to organize related formulas. In addition, we want to have a qualifying statement like *the set G together with the binary operation f, the inverse and the unit element is a group.* That is, we define at the same time a *predicate* which basically reads *X is a group* and has four arguments.

How can this be achieved? When we define a locale there has to be a mechanism that constructs the predicate according to the contents of the locale and assumes introduction and elimination rules. To illustrate this more closely, let us consider the hypothetical example of a locale for groups. The mechanism for the automatic generation of the first class representation, which we implemented experimentally, constructs a constant `group`.

```
group :: "['a set, ['a, 'a] => 'a, 'a => 'a, 'a] => prop"
```

This meta-level predicate can then be used to qualify parameters of appropriate type as a group as `PROP group G f inv e`. To integrate the associated meaning we generate and assume introduction and elimination rules according to the rules of the group locale[1].

```
groupI     "[| f ∈ G -> G -> G; inv ∈ G -> G; e ∈ G;
               ∀ x ∈ G. f (inv x) x = e; ∀ x ∈ G. f e x = x;
               ∀ x ∈ G. ∀ y ∈ G. ∀ z ∈ G. f (f x y) z = f x (f y z)
             |] ==> PROP group G f inv e"
groupE1    "PROP group G f inv e ==> f ∈ G -> G -> G"
```

[1]The leading `PROP` indicates Isabelle meta-level propositions.

```
groupE2    "PROP group G f inv e ==> inv ∈ G -> G"
groupE3    "PROP group G f inv e ==> e ∈ G"
groupE4    "PROP group G f inv e ==> ∀ x ∈ G. f (inv x) x = e"
groupE5    "PROP group G f inv e ==> ∀ x ∈ G. f e x = x
groupE6    "PROP group G f inv e ==>
      ∀ x ∈ G. ∀ y ∈ G. ∀ z ∈ G. (f (f x y) z = f (x) (f y z))"
```

The latter ones correspond to the rules of the locale but with the assumption of `group` made explicit as a premise. The first one is an introduction rule and the latter ones are elimination rules for the notion `group`.

4.7.1 Advantage for Operations on Locales

An explicit first class representation for locales has — apart from adequacy — advantages when we think about scoping, *i.e.* opening and closing and the outside view of the locale. Opening a locale means making its assumptions visible. As we have seen above, the way this is achieved is to generate meta-level assumptions `P [P]` for *all* locale rules `P`. Generating a predicate makes this process more efficient. Invocation of a locale corresponds then to the assumption of the predicate representing the locale. From this single assumption the locale rules are automatically generated by applying the elimination rules for the locale. The scope for locales merely has to keep track of the locale predicates of opened locales to administer the current context.

Theorems that are exported become theorems at the global level under the assumption of the locale predicate instead of listing all used rules separately as premises.

Multiple invocations can be realized by supplying distinct sets of parameter names to the invocation mechanism. For example, if we want to build up a context in which there are two different groups, we can open the locale for groups with two different sets of names. This results in a current context in which, say, `group G1 f1 inv1 e1` and `group G2 f2 inv2 e2` are stacked as assumptions. The opening mechanism generates different sets of invoked locale assumptions from the group elimination rules, *e.g.* `group_rule_G1` becomes `f1 ∈ G1 -> G1 -> G1` and `group_rule_G2` becomes assigned to `f2 ∈ G2 -> G2 -> G2`.

Instantiation of a locale corresponds to proving `group G f inv e` for some terms `G`, `f`, `inv` and `e` of appropriate type. In other words, instantiation of a locale can be implemented by application of the locale predicate. By backward resolution with the introduction rule of the locale, here `groupI`, this goal is unfolded to produce the corresponding subgoals according to the locale rules.

Although the explicit first class representation of a locale does have advantages we did not pursue the development of the mechanical generation of the predicate.

4.7.2 Disadvantages

It would have been nicer to define the meaning of the locale predicate by just one meta equality instead of a long list of elimination rules and an introduction rule.

```
"PROP group G f inv e ==  f ∈ G -> G -> G  ∧  inv ∈ G -> G
 ∧  e ∈ G  ∧  (∀ x ∈ G. f (inv x) x = e)
 ∧  (∀ x ∈ G. f e x = x)
 ∧  (∀ x ∈ G. ∀  y ∈ G. ∀ z ∈ G. f (f x y) z = f x (f y z))"
```

But, here we use a logical conjunction which is not part of the meta-logic of Isabelle. Isabelle implements just a fragment of higher order logic as its meta-logic (*cf.* Section 2.1). An extension of the Isabelle meta-logic with conjunction would be a possible way to enable simpler definition of the predicate in the above style.

Another problem with the representation of a locale by a meta-level predicate is that the predicate can have in general more than one argument, *e.g.* like in the above example four. It is necessary for a good formalization to pack such arguments in a product or some other structure. Then we can abstract from the argument list using just one variable. Otherwise, the argument lists get longer and longer when we consider higher order structures that have simpler structures as their arguments. An example for such structures are homomorphisms that have two groups as their parameters and would consequently have at least eight arguments. Unfortunately, the meta-logic of Isabelle does not have products either.

One might think about extending the meta-logic of Isabelle by conjunction and products. But, as we will see in the following chapter the adequate representation of a modular structure requires even more than those two features. Eventually, the extension of Isabelle's meta-logic would result in producing a fully equipped higher order logic as meta-logic — where we already have it as an object logic.

A way out of this dilemma is to restrict the locale concept to an object logic that offers sufficient support, say HOL. The object logic HOL contains a conjunction, so we can represent the predicate as above. Furthermore it has product types. In a further prototypical experiment we implemented the locale concept as a mechanism restricted to HOL. In this implementation the locale predicate is a unary HOL predicate. For our group example the locale mechanism generates

```
group :: "('a set * (['a, 'a] => 'a) * ('a => 'a) *'a) => bool"
```

We can now omit the PROP in all related rules and define the meaning of the locale by equality and conjunction. To enable access to the components of the product, our prototype automatically generates projection functions for the components of the group. We can integrate the predicate in any

other HOL formula giving us the first class property as we wanted it. Since the predicate is unary over the product type one variable can represent the arguments. Hence formalization of higher order structures becomes feasible. For example, we can write `group G` and thereby define even higher order structures, like homomorphisms, in a concise way.

But, Isabelle is a generic theorem prover and offers more object logics than just HOL. The concept of locales is more generally useful if not restricted to just one object logic. Furthermore, we do not always associate a logical predicate with a locale. For example, in the case study of Sylow, we can use the locale concept to assemble the assumptions of Sylow's proof. These assumptions and local definitions definitely define a context for a proof but they are not describing a structure.

Consequently, there is not sufficient reason to automatize the generation of a logical representation of a locale generally. In cases of local proof contexts, where we really think of those contexts as structures, a first class representation can still be defined individually by the user. In the following chapter we define a method for doing this in HOL schematically. The concept of locales, though, remains a global concept of local proof contexts for *all* of Isabelle logics and has no explicit representation as a predicate, neither in the meta-logic, nor in HOL.

4.8 Discussion

The syntax is strongly improved by locales because they enable dependent local definitions. Locale constants can have pretty printing syntax assigned to them and this syntax can be dependent as well, *i.e.* they can use everything that is declared as fixed implicitly. Thereby, locales approximate a natural mathematical style of formalization. Locales are a simpler concept than modules. They do not enable abstraction over types or type constructors. Neither do they support real schematic polymorphic constants and definitions as the full theory level does. They realize a sectioning device for grouping theorems together and sharing common assumptions. Since locales do not contain any type declarations it is in principle possible to generate a representation of a locale as a meta logical predicate. Through this representation, locales are first class citizens of the meta-logic. We developed this aspect of locales in an experiment described in Section 4.7. Although it worked well, the meta-level of Isabelle is not strong enough to realize a more adequate representation than predicates. An extension of Isabelle's meta-logic seems to drive the design out of proportion. Hence, we abandoned the automatic generation of an explicit first class representation for locales.

In some sense, however, locales do have a first class representation even as they are implemented now: globally interesting theorems that are proved in a locale may be exported. Then the former context structure of the locale

gets dissolved: the definitions become expanded (and thus vanish). The locale constants turn into variables, and the assumptions become individual premises of the exported theorem. Although this individual representation of theorems does not entail the locale itself as a first class citizen of the logic, the context structure of the locale is translated into the meta-logical structure of assumptions and theorems. In so far we reflect the local assumptions that constitute the locale into a representation in terms of Isabelle's meta-logic. This translation corresponds logically to an application of the introduction rule of the universal quantifier and the implication of the meta-logic. And, because Isabelle has a simple meta-logic this *quasi* first class representation is easy to apply.

Sometimes, premises that are available in a locale are not used at all in the proof of a theorem. In that case the exported version of the theorem will not contain these premises. This may seem a bit exotic, in that theorems proved in the same locale scope might have different premise lists. That is, theorems may generally just contain a subset of the locale assumptions in their premises. That takes away uniformity of theorems of a locale but grants that theorems proved in a locale can be individually considered for the export. In many cases one discovers that a theorem that one closely linked with, say, groups actually does not at all depend on a specific group property and is more generally valid. That is, locales filter the theorems to be of the most general form according to the locale assumptions. Hence, locales may also help to discover generality of theorems.

Locales are, as a concept, of general value for Isabelle independent of abstract algebraic proof. They are already applied in other theories of Isabelle, *e.g.* for reasoning about finite sets where the fixing of a function enhances the proof of properties of a "fold" functional. Locales are also used in proofs about multisets, the formal method UNITY, and the proof of the Ultrafilter Theorem [Fle99]. Furthermore, the concept can be transferred to all higher order logic theorem provers. There are only a few things the concept relies on. In particular, the features needed are implication and universal quantification — the two constructors that build the basis for the reflection of locales *via* export and are at the same time the explanation of the meaning of locales.

We believe that the improvement of concrete syntax is the most practical advantage of the locale concept. We found that it is better to separate concerns and to perform the first class reasoning separately in HOL using a construction of dependent types [Kam99]. We present this work in the following chapter.

Chapter 5

Modular Structures as Dependent Types

This chapter describes a method for representing algebraic structures in the theorem prover Isabelle. We use Isabelle's higher order logic. Dependent types, constructed as HOL sets, are used to represent modular structures by semantical embedding. The modules remain first class citizens of the logic. Hence, they enable adequate formalization of abstract algebraic structures and a natural proof style.

5.1 Introduction

The initial aim of our work was to find a module system for the theorem prover Isabelle where modules are first class citizens, *i.e.* have a representation in the logic. As we pointed out in Section 1.4, this is important when we want to formalize mathematical theories for abstract entities. The theorem provers we have inspected use their modules as representations for algebraic structures. Although the encapsulation and abstraction achieved by packaging structures into modules is sensible, it does not constitute an adequate representation.

In the previous chapter we introduced locales. They cover the locality and scoping aspects of modules. As we have seen there, the export facility automatically produces a meta-level formula of a theorem that can be considered as a first class representation of the locale. Generally one can say that a locale *has* a first class representation; we can write down the locale as a term. This is because we abandon the use of type declarations in locales. We even considered the possibilities for an automatic generation of a first class representation. Nevertheless, it turned out that the structural concepts to represent a locale — and in general a module — adequately, go well beyond the expressivity of Isabelle's meta-logic. In the present chapter we want to investigate a first class representation of modular structures by

starting from abstract algebraic structures. We choose the object logic HOL, which is the implementation of higher order logic we already discovered to be suitable for abstract algebraic reasoning in Chapter 3. Using an extension of Isabelle/HOL with a notion of sets, we define dependent types to find an embedding of signatures in the logic. Using this embedding, we can represent abstract algebraic structures as dependent types. Furthermore, we use record types [NW98] to represent the element patterns of algebraic structures. This changes our representation slightly; for example, we represent a group as a record with four fields: the carrier set, the binary operation, the inverse, and the unit element. The class of all groups is represented by a HOL set over this record type.

In this chapter, we first explain our notion of algebraic structures and give examples in Section 5.2. In Section 5.3 dependent types and their formalization in Isabelle/HOL are introduced. Their application to represent structures is described. Finally, we discuss some related work and draw some conclusions in Section 5.4.

5.2 Algebraic Structures

An algebraic structure is a class of entities, which are considered to be similar according to some characterizing rules, while abstracting from other concrete characteristics. Examples for algebraic structures are groups, rings, homomorphisms, *etc.* The algebraic structure of groups, say, is the class of all objects that satisfy the group axioms. Hence, the structure is formed by abstracting over elements of similar appearance that fulfill common properties.

In this section we characterize our notion of simple algebraic structure and higher order structure. We use an informal notion of signature instead of modules because that is what the latter basically are. We do not use a separate syntactical description language for those signatures, because we think that for the encoding of mathematical structures our method of direct encoding in HOL sets plus dependent types is sufficiently self-explanatory.

In the following we will talk about structures as sets of objects. We are using the set notion of Isabelle/HOL as a foundation for this work. This notion is defined in terms of predicates and is thus — in a set-theoretic sense — rather a notion of *classes* than sets.

5.2.1 Simple and Higher Order Structures

An algebraic structure is a class of mathematical objects. They can be syntactically represented by their signature, *i.e.* by the arities of their elements and the rules which hold for the elements of the structure. An object matching the arities and fulfilling the rules is an element of a structure. A

syntactical description of a structure S by its signature and related rules is of the form:

$$\begin{array}{ll}\textbf{signature} & S(x_1,\ldots,x_n)\\ & x_1 \in A_1\\ & \ldots\\ & x_n \in A_n\\ & P_1\\ & \ldots\\ & P_m\end{array}$$

where the A_i are the arities of the parameters, $i \in \{1,\ldots,n\}$. The P_k are properties in which the parameters x_i can occur, $k \in \{1,\ldots,m\}$. The arities can denote types or sets depending on the framework. This syntax is a simplified form of the style of modules as seen in other theorem provers [OSRSC98, FGT93, GH93].

The associated meaning of this syntactical description of signature S is what we consider as an algebraic structure

$$[\![S]\!] \equiv \{(x_1,\ldots,x_n) \in A_1 \times \ldots \times A_n \mid P_1 \ \wedge \ \ldots \ \wedge \ P_m\}$$

where in P_i any of $\{x_1,\ldots,x_n\}$ can occur. We call the elements $x_1,\ldots,x_n$ *parameters* of the structure S.

Structures may possibly be parameterized over other structures. We call such structures *higher order structures* in contrast to *simple structures.* To identify the structures that are parameters of higher order structures, we use the term *parameter structures*, and the structure that is defined by the higher order structure itself we call *image structure.*

For the definition of simple structures, we use sets of extensible records. Record types are used as a template for the structure's elements. They give us the selectors, which are projection functions enabling reference to the constituents of a simple structure.

The definition of higher order structures needs a device to refer to the formal parameters. Here we employ the set theoretic construction of dependent types. It enables the use of constraints on parameter structures in the definition of an image structure. The selectors of the parameter structures admit to refer to their constituents.

For the parameter tuple $par \equiv (x_1,\ldots,x_n)$ of a simple structure we define a record α *par-sig* as[1]

$$\begin{array}{lll}\textbf{record} & \alpha \textit{ par-sig} & \equiv\\ \_.\langle x_1\rangle & :: A_1 & \text{(postfix)}\\ \ldots & \ldots & \ldots\\ \_.\langle x_n\rangle & :: A_n & \text{(postfix)}\end{array}$$

[1]We assume here syntax definition possibilities that are planned for records though not yet available. They can be modelled using usual syntax declarations, though.

The underscore defines the argument positions of the field selectors of this record. For example, if T is a term of appropriate record type, *i.e.* a suitable n-tuple, we can select the field x_j of T by $T.\langle x_j \rangle$. In general, the elements of the tuple will have names like *carrier*, or *inverse*, so the naming discipline of the record field selectors that we chose, is more informative than indexing by numbers.

The representation of a simple structure is given as a set of records; the record type defines the element pattern of the structure.

5.2.2 Example: Groups and Homomorphisms

A *group* is constituted by a carrier set and a binary function $\circ$ on that set, such that the function $\circ$ is associative, and for every element x in the carrier there exists an inverse x. The carrier set also contains a neutral element e. The syntactical representation of a group by a signature is

signature		Group $(cr, \circ, inv, e)$
$\circ$	$\in$	$cr \times cr \to cr$
inv	$\in$	$cr \to cr$
e	$\in$	cr
$\forall\ x \in cr.$		$e \circ x = x$
$\forall\ x \in cr.$		$inv(x) \circ x = e$
$\forall x, y, z \in cr.$		$(x \circ y) \circ z = x \circ (y \circ z)$

According to Section 5.2.1, the mathematical meaning that we associate to this example is

$$[\![\ \text{Group}\ ]\!] \equiv \{(\!|\ cr, \circ, inv, e\ |\!) \mid \begin{array}{l} \circ \in cr \times cr \to cr\ \wedge\ inv \in cr \to cr\ \wedge\ e \in cr\ \wedge \\ (\forall x \in cr.\ e \circ x = x)\ \wedge \\ (\forall x \in cr.\ inv(x) \circ x = e)\ \wedge \\ (\forall x, y, z \in cr.\ (x \circ y) \circ z = x \circ (y \circ z))\} \end{array}$$

The notation $(\!|\ cr, \circ, inv, e\ |\!)$ of the elements of this set stands for an extensible record term. In this context it is sufficient to understand them as products. The base type of the set Group is defined by the following extensible record definition.

record	α *group-sig*	$\equiv$
$\_.\langle cr \rangle$	$::\ \alpha$ *set*	(postfix)
$\_.\langle f \rangle$	$::\ [\alpha, \alpha] \Rightarrow \alpha$	(postfix)
$\_.\langle inv \rangle$	$::\ \alpha \Rightarrow \alpha$	(postfix)
$\_.\langle e \rangle$	$::\ \alpha$	(postfix)

The structure Group is of type (α *group-sig*) *set*. In the following example of a higher order structure for group homomorphisms we see how the field selectors are used to refer to the constituents of a group. Subsequently, we name the group operation f, instead of $\circ$, because we need to refer to the

group G and in prefix notation $G.\langle f\rangle$ looks more natural. An improvement of syntax for such implicit references is given by the locale concept we have presented in the previous chapter.

A homomorphism of groups is a map from one group to another group that respects group operations. The parameters of a structure Hom for homomorphisms are groups themselves, *i.e.* we have a higher order structure. The following syntactical form encloses the parameter structures in square brackets.

$$\begin{array}{rl} \textbf{signature} & \text{Hom } [\ G, H \in \text{Group }]\ (\ \Phi) \\ \Phi \in & G.\langle cr\rangle \to H.\langle cr\rangle \\ \forall x, y \in & G.\langle cr\rangle.\ \Phi(G.\langle f\rangle\ x\ y) = H.\langle f\rangle\ \Phi(x)\ \Phi(y) \end{array}$$

In the definition of the mathematical structure we have to add "where G and H are elements of the structure *Group*". That is, a mathematical object representing a homomorphism between groups has to carry also the two groups in itself. It is a triple (G, H, Φ) of two groups and a homomorphism between them. The definition of homomorphism uses the elements G and H. So we need to refer to the parameter structures G and H when we define the image structure; the definition of the image structure depends on the parameter structures. Hence, we choose a dependent type, the Σ-type, to define the structure for homomorphisms.

$$\begin{array}{rl} Hom \equiv & \Sigma_{G \in Group}\ \Sigma_{H \in Group} \\ & \{\Phi \mid \Phi \in G.\langle cr\rangle \to H.\langle cr\rangle \wedge \\ & (\forall x, y \in G.\langle cr\rangle.\ \Phi(G.\langle f\rangle\ x\ y) = H.\langle f\rangle\ \Phi(x)\ \Phi(y))\} \end{array}$$

Now the parameter groups G and H are bound by the Σ operator and we can refer to them, and their constituents by using the projections, *e.g.* $G.\langle f\rangle$. In the following section, we explain the notion of dependent types, and how we represent them using set theoretic constructions.

5.3 Dependent Types as Structure Representation

According to a textbook introduction to type theory [NPS90, page 52] the main reason for the introduction of the Π-set is the interpretation of the universal quantifier. The Heyting interpretation of this quantifier is [Hey56]

> $\forall x \in A.B(x)$ is true if we can construct a function which when applied to an element a in the set A, yields a proof of $B(a)$.

However, the Π-construction is also well known in set-theory (*e.g.* [Hal60]). The dependent sum Σ enables to deal with the existential quantifier, *i.e.* $\exists$ can be understood constructively as

$$\exists x \in A.B(x) \equiv \Sigma_{x \in A} B(x)$$

We use the dependent sets in the same sense, but restrict the use to the description of structures. We consider A and B as structures, not general formulas. So, we use the dependent sets as type theory uses them, but in a more naïve way restricting ourselves to the statements $x \in A$ and *not* interpreting this as "x is a proof of formula A". We interpret the dependent types as modular structures; Σ as relations and Π as function sets.

The idea is to use the syntactical signature description of the structure as a set $B(x)$ — with a formal parameter x. This formal parameter is an element of the first set A. In case of more than one parameter structure the nesting of the dependent type constructors Σ and Π just accumulates.

We show in this section how dependent types are formalized in HOL, and how this formalization can be used to represent higher order structures.

5.3.1 Isabelle Representation

Isabelle/HOL implements a simple type theory [Chu40] and has no dependent types. It is extended by a notion of sets. Sets are here essentially predicates, rather than "built-in" by ZF-style axioms. We use this extension to define dependent types as sets in Isabelle.

Set Representation

One can consider the Σ-type as a general form of the Cartesian Product. If we represent $\Sigma_{x \in A} B(x)$ as a set, it is thus

$$\Sigma_{x \in A} B(x) \equiv \bigcup_{x \in A} \bigcup_{y \in B(x)} \{(x, y)\}$$

This representation of the Σ-type is used in HOL.

The Π-type is the type of dependent functions. It is related to the Σ-type. We can express this type as a set by considering the subsets of Σ which can be seen as functions

$$\Pi_{x \in A} B(x) \equiv \{f \in \mathcal{P}(\Sigma_{x \in A} B(x)) \mid \forall x \in A.\, \exists!\, y \in B(x).\, f(x) = y\} \qquad (5.1)$$

where $\mathcal{P}$ denotes the powerset.

Implementation in Isabelle

In the distribution of Isabelle/HOL the Σ-type is already defined in terms of HOL sets, the Π-type not.

The most natural way to define Π seems to be to use Definition (5.1) defining Π in terms of Σ. But, then the functions we would get would be sets of pairs and we would develop a new domain of functions inside HOL, when there are already functions.

The existing functions in HOL are the elements of the function type $\alpha \Rightarrow \beta$, where α and β indicate arbitrary types, and $\Rightarrow$ is the function type constructor. There is a notation for λ-abstraction available to express functions. We would like to define function sets, *i.e.* sets of elements of the HOL type $\alpha \Rightarrow \beta$, and on top of that we want to have that the co-domain of these functions β may depend on the input to the function. Ideally, the type β should depend on some x of type α. Since HOL does not have dependent types, it is impossible to integrate the dependency at the level of types. But, we can define a non-dependent type for the constructor Π as

$$[\alpha\ set, \alpha \Rightarrow \beta set] \Rightarrow (\alpha \Rightarrow \beta) set$$

Then, we can assign the above type to a constant Π in HOL and add the idea of dependency to the definition of this constructor.

$$\Pi_{x \in A} B(x) \equiv \{f \mid \forall x.\ \text{if } x \in A \text{ then } f(x) \in B(x) \text{ else } f(x) = (\epsilon y.\text{True})\}$$

By using the more explicit language of sets we achieve that the co-domain is a set which depends on the argument to the function. The "else" case is necessary to achieve extensionality for the Π-sets. The ϵ is the Hilbert-operator for HOL that we have used before (*cf.* Chapter 3).

Equality compares functions according to their behavior on the set A. We do not care about what a function in $\Pi_{x \in A} B(x)$ does outside A. We want to think of all functions which behave alike on A as the same function. Obviously, this is not generally true from the perspective of HOL. That is why we have to label one distinguished element out of the set of all functions which behave equally on A by determining the value for inputs outside A to the arbitrary value $(\epsilon y.\text{True})$. In the classical HOL system [GM93] this term is called `ARBITRARY`. It suffices our purposes, but we have to admit that it is not a fully satisfying solution. What we really want to have here is some distinct value, like the undefinedness $\perp$ of domain theory that would be a member of all sets — as $\perp$ is an element of all domains.

The dissatisfaction in the above definition stems from the fact that we do not know if $(\epsilon y.\text{True})$ is in $B(x)$ or not. Thus, we cannot decide how any function behaves if applied to this value. This causes problems when we think about composition, for example, but they can be overcome by defining corresponding restrictions, which transform a function into the appropriate form of the Π-set. We define this restriction according to the similar formalization in ZF by using a λ-notation, so that we can now annotate the HOL function f restricted to a set A as $\lambda x \in A.fx$.

The non-dependent function sets are a special case of the definition of Π. Using Isabelle's pretty printing facilities, we get a nice syntactical representation for them and can now write $A \rightarrow B$ for the set of functions from a set A to a set B.

To reassure ourselves that the definition of Π is sound we have established a bijection Π_{Bij} between the classical definition from Equation 5.1 and the above HOL function set as

$$\Pi_{Bij} AB \equiv \lambda f \in \Pi_{x \in A} B.\ \{(x, y) \mid x \in A \ \wedge \ y = fx\}$$

We proved in Isabelle that this map is actually a bijection.

Our approach to model partial functions in a total setting is in principle similar to the one described in [FFL97]. There, the membership of the argument to the function is explicitly modelled by a notion of domain that is carried along as additional assumption. Although our approach does not enable an explicit reasoning about the domain of a function, the implicitness grants that we can use the usual function application and do not have to integrate the domain in all our formulas. In case of actual application of one of our λ-functions, the actual proof obligation, *i.e.* the proof that the argument is in the domain, is exactly the same as in the other approach with explicit domains.

5.3.2 Algebraic Formalization with Π and Σ

We concentrate in this section on the representation of higher order structures. As already pointed out in Section 5.2, we use sets of records for simple structures. We use the dependent type constructors Σ and Π to represent higher order structures, that is, to express structures where the parameters are elements of structures themselves. Roughly speaking, the Σ-types are used for general relations between parameter and image structures. When this relation is a function, *i.e.* the construction of the image structure is unique and defined for all elements of the parameter structure, then we can construct elements of the higher order structure using the λ-notation. In that case, the higher order structure is a set of functions, *i.e.* it is a Π-type structure.

Use of Σ

The interpretation of the Σ-type is that of a relation between parameter and image structure. Higher order structures whose image structures are defined for certain input parameters, but not necessarily for all, can be represented by Σ. So, the elements of these higher order structures are pairs of parameter and image structure elements; for a structure $Struc \equiv \Sigma_{x \in A} B(x)$, we can write this membership as $(a, b) \in Struc$.

But in addition to expressing the membership of a pair of parameter and image structure elements in the higher order structure, we also want to instantiate the higher order structure with an element of the parameter structure. Instantiation of a higher order structure corresponds to application of the structure when we see it in a functional way. By $Struc \downarrow a$

we annotate the instantiation or application of the structure. What we are interested in is to get an instance of the image structure B, where a is substituted for the formal parameter x. That is, we want to derive $B(a)$ for $a \in A$, or apply the entire structure generally to an element of the parameter structure. For $a \in A$ we construct an operator $\downarrow$ such that $(\Sigma_{x \in A} B(x)) \downarrow a$ evaluates to $B(a)$. We can define $\downarrow$ in terms of the image of a relation, so it reduces to[2]

$$Struc \downarrow a \equiv (\Sigma_{x \in A} B(x)) \hat{} \hat{} (\{a\}) \equiv \{y \mid \exists x \in \{a\}.(x, y) \in \Sigma_{x \in A} B(x)\}$$

Then we can use the theorem

$$(a, b) \in \Sigma_{x \in A} B(x) \Rightarrow b \in B(a)$$

to derive

$$a \in A \Rightarrow Struc \downarrow a = B(a)$$

This theorem enables us now to build the instance of a higher order structure with an element a of the parameter structure A.

Use of Π

Elements of higher order structures which are uniquely defined — like the factorization of a group in Section 6.3.1 — can be represented by a function definition in the typed λ-calculus from Section 5.3.1. Let $elem \equiv (\lambda x \in A.\ t(x))$, then

$$elem \in \Pi_{x \in A} B(x) \ \text{ iff } \ \forall a \in A.\ t(a) \in B(a)$$

The function body t of the element *elem* constructs elements of the image structure of the higher order structure to which *elem* belongs. For the application or instantiation we do not need an extra operator as $\downarrow$ for Σ. Since we have defined the Π-type as sets of functions we can use the HOL function application $elem(a)$. If $a \in A$ then this evaluates to $t(a)$. We have to apply reasoning to evaluate the definition of λ in such a reduction. But this can be solved by the simplifier.

The λ-functions we construct in this way are elements of a Π-set. The domain of the Π-set that contains a λ-function is the same as the domain of the λ-function. The co-domain can be any set that includes the image of the λ-function. In the example of factorization of a group, or direct product of a group in Section 6.3.1, we will see that this image structure is again the structure of groups. The higher order structure is a Π-set; with λ we construct only *elements* of the structure. The membership proof, *i.e.* the proof that a λ-function is in a Π-set, is a process very similar to typing. This typing can mean a logical judgement, depending on the higher order

[2]The Isabelle notation for the operator representing the image of a relation is ^^.

structure, to which we want to assign a λ-function to through this typing. Again, in the example of factorization and direct product of groups in Section 6.3.1 this typing process corresponds to showing that the constructed images of the λ-terms are groups. Hence, the logical meaning of this typing is that the constructed image is a group. The assignment of a type in the described way is naturally not unique. On the one hand, this reflects the property of subtypes, and that elements of a subtype are also elements of the supertype. On the other hand, it corresponds to the implication of logical properties.

This relationship between types and propositions resembles the Curry-Howard isomorphism [How80] between types and propositions. It is very explicit, because the type representations are sets. Hence, we can immediately apply the isomorphism by using the axiom of collection, *i.e.*

$$a \in \{x \mid P(x)\} \Rightarrow P(a)$$

where $\in$ corresponds to the typing notation ":". The difference in our approach to the type paradigm of Curry and Howard is that we do not use the correspondence as a basis of the logic, so that logical formulas *do not have* to be represented by a type. In type theory the existential quantifier *is* a Σ-type, *i.e.* the type is the only encoding of logic. The Curry-Howard isomorphism is explicitly present in our approach, since we have both representations — sets and propositions. The actual isomorphism is represented by the axiom of collection.

However, the correspondence between Π-set structures and imposing a universal proposition over a construction may also lead to an alternative method of representing higher order structures. In the following section we want to describe briefly, how we can use Π instead of Σ to impose a certain typing discipline. We explore this alternative way only to give a complete description of dependent types as sets, not as the way we suggest to use them as structure representation.

Alternative Use of Π

In principle one can define structures that are universally applicable to parameters directly by $\Pi_{x \in A} B(x)$. For example, we may use Π instead of Σ to encode the structure of group homomorphisms, because for all groups G and H there is always a homomorphism between G and H. The use of Π as general representation for higher order structures in the described sense is more complicated than Σ.

In case of a higher order structure where we know that there exists always an image structure for each parameter structure, *i.e.* if we have a set of functions, the structures can be defined using Π instead of Σ. We can use the theorem

$$a \in A \wedge f \in \Pi_{x \in A} Bx \Rightarrow f(a) \in B(a)$$

to apply the functions individually. If

$$\forall x \in A.\exists y.\ y \in B(x)$$

holds, we can establish

$$\Pi_{x \in A} B(x) \neq \varnothing \tag{5.2}$$

which is equivalent to $\exists f.\ f \in \Pi_{x \in A} B(x)$. In this case only we can build the instance of the structure. Similar to the construction in Section 5.3.2 we define an instance operator $\Downarrow$. As the instance operator for Σ, it uses the image operator for a relation.

$$F \Downarrow a \equiv (\lambda f.fa)^{\frown\frown}(F)$$

The definition of this operator enables together with Equation 5.2 the construction of the general instantiation of a structure defined in terms of Π.

$$Struc = \Pi_{x \in A} B(x) \wedge x \in A \wedge Struc \neq \varnothing \Rightarrow Struc \Downarrow a = B(a)$$

5.3.3 Relationship between Π and Σ

It is an extra effort to use the Π-representation for a structure; we have to show the nonemptiness of the set $\Pi_{x \in A} B(x)$. The reason for this extra work is that the Π-representation entails

$$f \in \Pi_{x \in A} B(x) \Rightarrow \forall x \in A.f(x) \in B(x) \tag{5.3}$$

The antecedent is equivalent to $\forall x \in A.\ \exists y.\ y \in B(x)$. It implies that all $B(a)$ for all $a \in A$ are nonempty. To keep things simple it is more sensible to use generally Σ, even if Π would be applicable. We avoid the extra effort of justifying Property (5.3) for a structure by showing Condition (5.2). In the following chapter we show a list of examples from abstract algebra where we use Σ for structure representation. We can replace all Σ by Π and still get the same theorems. This is the case because in the examples of higher order structures in abstract algebra, there seems to be a functional relationship between the parameter and the image structure. This is sometimes called *well-definedness* of mathematical definitions.

We explain the relationship between the representation by Σ and Π again with the example of group homomorphisms. These maps can be defined either as

$$\begin{aligned} Hom_{\Sigma} \equiv \quad & \Sigma_{G \in Group}\ \Sigma_{H \in Group} \\ & \{\phi \mid \phi \in G.\langle cr \rangle \rightarrow H.\langle cr \rangle \wedge \\ & \forall x, y \in G.\langle cr \rangle.\ \phi(G.\langle f \rangle\ x\ y) = H.\langle f \rangle\ \phi(x)\ \phi(y)\} \end{aligned}$$

or, using Σ instead, as

$$\begin{aligned} Hom_{\Pi} \equiv \quad & \Pi_{G \in Group}\ \Pi_{H \in Group} \\ & \{\phi \mid \phi \in G.\langle cr \rangle \rightarrow H.\langle cr \rangle \wedge \\ & \forall x, y \in G.\langle cr \rangle.\ \phi(G.\langle f \rangle\ x\ y) = H.\langle f \rangle\ \phi(x)\ \phi(y)\} \end{aligned}$$

For example, the λ-function

$$\lambda\ G \in Group.\ \lambda\ H \in Group.\ \lambda\ x \in (G.\langle cr\rangle).\ (H.\langle e\rangle)$$

is a member of the set Hom_Π. This statement is equivalent to

$$(G, H, \lambda\ x \in (G.\langle cr\rangle).\ (H.\langle e\rangle))$$

is an element of Hom_Σ *for all* groups G and H. The former function in Hom_Π is represented by a set of triplets in Hom_Σ. If one wants to see the functions in Hom_Π represented as sets of pairs, its elements would be sets of subsets of Hom_Σ, or elements of $\mathcal{P}(Hom_\Sigma)$, respectively. So, one can understand Hom_Σ as the same as Hom_Π, but with the internal function structure dissolved.

It seems to us that the examples of higher order structures in abstract algebra are all capable of being defined in terms of Π, because they are all well defined. That is, the mathematical definitions already entail that they are meaningful, *i.e.* define nonempty image structures, and are thus sets of functions. Nevertheless we are forced to show this, if we employ Π instead of the weaker but simpler structure notation by Σ. This reflects the proof of well definedness of a new definition in a mathematical development.

Do we have any advantages, if we use Π instead of Σ to represent a higher order structure? By showing the nonemptiness or well definedness of a structure in advance, we incorporate this property in the definition and can reveal it in contexts where it is needed. On the other hand, if it is really needed[3] we could as well show it for the equivalent Σ version of the structure. One might argue that the use of Π forces a certain discipline to define only nonempty, *i.e.* well defined, higher order structures.

Summarizing, the way we use Σ and Π is as follows:

- We express higher order structures that are uniquely defined for their parameter structures by λ-expressions. These structures are then members of corresponding Π-sets. We will illustrate this by examples in Section 6.3.1 and 6.3.2.
- We use Σ to describe all other higher order structures. That is, higher order structures where we do not have an explicit method to construct image elements explicitly from parameter elements. These are the cases where there is no functional relationship required, *i.e.* the structures are relations between parameter and image structure.

5.4 Discussion

The approach to use dependent types as modular structures is well known (*e.g.* [Mac86]), but to represent these types as sets is a new way of mechanizing modules. The advantage lies in the first class property of the structures.

[3]Non of our examples uses this property of a Π structure.

As we have seen in Section 5.3.2, we can now define operations on modular structures in the logic. Others, like forgetful functors and unions of structures can be expressed in terms of those. This will be illustrated in the following chapter.

There has been work done on integrating dependent types into higher order logic [JM93]. The objectives of this work are similar to ours in that the focus is on obtaining the "expressive advantage". The translation of dependent types into HOL constructed there, corresponds to "sending a dependent type to a predicate". Nevertheless, the realization is quite different in that it aims at constructing dependent types as ML abstract data types. This is in the tradition of LCF. Our approach taken here is much shallower in that we just express the dependent types as sets. Still, we gain the first class property of structures because they are terms of the logic and thereby we get the expressive advantage as well.

The HOL system [GM93] has a concept of abstract theories based on the experiments with abstract algebra [Gun89]. Gunter presented the ideas at the Third HOL Users Meeting [Gun90]. Windley implemented them later in HOL [Win93]. Theories are there represented by two components: abstract representations and theory obligations — roughly resembling our notion of parameter and image structures. In contrast to our work the obligations are represented by predicates over polymorphic types. The base sets of structures are types, not sets as in our representation.

Although our dependent types are realized by an embedding it is not too deep[4]. That is, it keeps a balance between the extremes of internalizing everything within the logic as first-class (resulting in unwieldy concepts), or externalizing too much (becoming less expressive).

5.4.1 Relation to Type Theory

LEGO [Bur90, LP92], an implementation of the Extended Calculus of Constructions [Luo90b] uses dependent types as theories [Luo90a]. Our work is similar except that we are constructing these types as sets.

One difference is that our dependent structures are terms, not types as in ECC. The discussion section of [Luo90a, Section 4.4] mentions the possibility of a combination of two ideas: one is to have dependent structures as a representation of theories, done by ECC. The other idea is to have operations on theories, that is theories are values and there are operations that can be performed on those values. These concepts were examined in the specification language CLEAR [SB83]. Since the dependent types are values in our semantical embedding of theory structures, it is possible to define operations on theories as HOL functions. This has been illustrated

[4]In this context we mean by deep, how explicitly we can reason about the subject of interest, *i.e.* modules

in this chapter by defining the operation of instantiation by the structure instance operation (*cf.* Section 5.3.2).

The other difference is that we are following the HOL philosophy of not considering proof objects. Thus the actual proof construction leading to the results is independent of the type structure of the formalization. Nevertheless, the structures we use contain enough information to produce the instances one is interested in, as will be illustrated by the proof for the group of ring automorphisms in Section 6.3.4.

5.4.2 Records

In an earlier version of this work we used products as base type for simple structures. Due to a suggestion of P. Martin-Löf at the TYPES 98 workshop [Typ98] we employ here extensible records, which were first presented at the workshop. Although Martin-Löf suggested to use records as an alternative to HOL sets, we use the concept of records just in addition to sets: records allow us to use their field selectors that are generated according to our needs. This is handy, but forces a different treatment of higher order structures. Higher order structures are built by using Σ-types, which are a sophisticated form of products. We adopted this heterogeneous representation of simple and higher order structure, because the records save some work. They automatically provide the projection functions. In Section 6.4, when we consider operations on theories, we will see that this incongruous representation works, though.

Naraschewski *et al.* suggest using extensible records for a representation of groups as follows (assuming the definition of monoids [NW98, Chapter 2]):

$$
\begin{array}{ll}
\textbf{record} & \alpha\ \textit{group-sig} \equiv \textit{monoid-sig} + \\
\quad \text{inv} & :: \alpha \Rightarrow \alpha \\
\textbf{defs} & \\
\quad \text{group} & :: (\!| \circ :: \alpha \times \alpha \Rightarrow \alpha, 1 :: \alpha, \textit{inv} :: \alpha \Rightarrow \alpha |\!) \Rightarrow \textit{bool} \\
\quad \text{group} & (\!| \circ, 1, \textit{inv} |\!) \equiv \textit{monoid}\ (\!| \circ, 1, \textit{inv} |\!) \wedge \forall x.\ (\textit{inv}\ x) \circ x = 1
\end{array}
$$

The idea of representing the base set of a group by a type does not constitute an adequate formalization of abstract algebra in Isabelle. This is because any property apart from internal group properties can only be expressed using formulas over types. For example, to express the subgroup property one needs subtypes and for factorization of groups one needs quotient types. All these advanced type concepts are not part of a simple theory of types as Isabelle and HOL are based upon — even when records are added.

The way we use records to describe base types of sets is closer to a notion of *dependent records.* The dependency between the elements of the record can be entailed in the description of the set representing the structure. To elaborate the notion of dependent records types fully, it would be necessary to embed them as usual products are embedded in HOL, so that there

would be a dependent record constructor similar to Σ. We consider this as an interesting line of research for the development of records in Isabelle.

5.4.3 Syntax

The syntax definition possibilities provided by locales (*cf.* Chapter 4) can improve the presentation of structures. For example terms like $G.\langle f\rangle\ x\ y$ should be expressible as $x \circ y$. This is nontrivial because the reference to the element G is crucial. Nevertheless, locales enable definitions depending on local assumptions. The additional use of locales with the structural concepts presented in this chapter provides a satisfactory style of abstract algebraic reasoning.

In the following chapter, we will evaluate the concepts we developed by an illustration how they can be used in combination to improve abstract algebraic reasoning. That is, we will reconsider the Sylow proof and do some more examples from abstract algebra using locales and the mechanization of modules as presented in this chapter.

Chapter 6

Locales + Dependent Types = Modules

In this chapter, we demonstrate our results. We have developed the concept of locales to realize local scoping and syntax. Furthermore, we have mechanized a first class representation of modules using dependent types. The main idea of this work is that a combination of these two concepts enables adequate representation and convenient proof in abstract algebra. To validate this hypothesis we present some application examples. We start off with the development of a basis for abstract algebra, *i.e.* we present groups and cosets. We continue in Section 6.2 with reconsidering the proof of Sylow's theorem and point out the improvements we gain from our conceptual developments. In the following sections we present some more new case studies: the formalization and proof of the group of bijections, ring automorphisms and the factorization of a group by a normal subgroup in Section 6.3, followed by a mechanization of the full version of Tarski's fixed point theorem in Section 6.3.5. Finally, we give in Section 6.4 a few examples for operations on structures.

6.1 Basic Formalizations

6.1.1 Groups and Subgroups

The formalization of groups in Isabelle has already been introduced in Chapter 3. In Chapter 5 we adapted this formalization of groups to our notion of simple structures; the base type of the set `Group` became a record. Thereby, we get the projection functions for free.

Similarly, we must update now the definition of subgroups introduced in Chapter 3 according to the general approach of structures as dependent sets. Note that compared to the definition of subgroups in Section 3.2.3, the restriction of the group operations to the base set H of the subgroup is

realized using our λ-notation. Subgroups are now the higher order structure **subgroup**. That is, the Σ-structure can be used to model the subgroup relation.

```
Σ G ∈ Group. {H. H ⊆ carrier G &
   ⦇ carrier = H, bin_op = λ x ∈ H. λ y ∈ H. (G.<f>) x y,
     inverse = λ x ∈ H. (G.<inv>) x, unit = (G.<e>) ⦈ ∈ Group)}
```

Although this change of the definition obviously affects the mechanized proofs we already constructed for subgroups in the Sylow case study, the changes to the proofs are minimal. Since we always derive introduction and elimination rules for new structures, we do so as well for subgroups. Hence, to adapt the proofs to the change of the subgroup property, only the derivations of these interface rules for the subgroup property have to be updated. Even the convenient syntax `H <<= G`, for H is a subgroup of G, can be preserved. It now abbreviates `(G, H) ∈ subgroup`.

The proofs for groups established as a basis for the subsequent case studies subsume the ones that we already proved for Sylow's theorem and extend them by some more specific results. In contrast to the derivations there, we use locales. We define a locale **group** to provide a local proof context for group related proofs (*cf.* Section 4.3).

```
locale group =
  fixes
    G        :: "'a grouptype"
    e        :: "'a"
    binop    :: "'a => 'a => 'a"    (infixr "#" 80)
    inv      :: "'a => 'a"          ("i (_)"      [90]91)
  assumes
    Group_G   "G ∈ Group"
  defines
    e_def     "e == (G.<e>)"
    binop_def "x # y == (G.<f>) x y"
    inv_def   "i x == (G.<inv>) x"
```

This locale is attached to the theory file for groups. Prior to starting the proofs concerning groups, we open this locale and can subsequently use the syntax and the local assumption `G ∈ Group` throughout all proofs for groups. This improves the readability of the derivations as well as it reduces the length of the proofs. For example, instead of

```
[| G ∈ Group; x ∈ (G.<cr>); (G.<f>) x x = x |] ==> x = (G.<e>)
```

we can state this theorem now as

```
[| x ∈ (G.<cr>); x # x = x |] ==> x = e
```

Subgoals of the form `G ∈ Group` that were created during the old proofs are not there any more because they are now matched by the corresponding locale rule. All group related proofs share this assumption. Thus, the use of a locale rule reduces the length of the proofs.

6.1.2 Cosets

We extend the set of theorems that we have already derived for Sylow's theorem in particular with respect to left cosets, a product, and an inverse operation for subsets of groups. We restructure the developments by creating a separate theory for cosets named `Coset`. We add the following constructors to the theory of cosets[1]

```
l_coset G a H == ((G.<f>) a) `` H
set_prod G H1 H2 == (λ (x, y). (G.<f>) x y) `` (H1 × H2)
set_inv G H == (G.<inv>) `` H
```

The definition of left cosets immediately gives rise to the definition of a special class of subgroups, the so-called *normal subgroups* of a group.

```
Normal ==  Σ G ∈ Group.
   {H. H <<= G & (∀ x ∈ (G.<cr>). r_coset G H x = l_coset G x H)}
```

We define the convenient syntax `H <| G` for `(G, H) ∈ Normal`. As is apparent from the definition, normal subgroups are a special case of subgroups of a group where left and right cosets coincide. This is not necessarily the case in non-abelian groups. Note that the use of the verbose syntax for cosets is only necessary in the very first mention of normal subgroups. We define a locale for the use of cosets to enable convenient syntax for cosets and products. This locale is defined as an extension of the locale for groups.

```
locale coset = group +
  fixes
    rcos     :: "['a set, 'a] => 'a set"     ("_ #> _" [60,61]60)
    lcos     :: "['a, 'a set] => 'a set"     ("_ <# _" [60,61]60)
    setprod  :: "['a set, 'a set] => 'a set" ("_ <#> _"[60,61]60)
    setinv   :: "'a set => 'a set"           ("I(_)"   [90]91)
    setrcos  :: "'a set => 'a set set"       ("{* _ *}"[90]91)
  defines
    rcos_def    "H #> x == r_coset G H x"
    lcos_def    "x <# H == l_coset G x H"
    setprod_def "H1 <#> H2 == set_prod G H1 H2"
    setinv_def  "I(H)  == set_inv G H"
    setrcos_def "{* H *} == set_r_cos G H"
```

The locale for cosets adds to the one for groups that we defined in the theory `Group.thy`. To open it, the scope representing the current proof context has to contain the locale `group`. If this is not the case, the function `Open_locale` will open it automatically (*cf.* Section 4.3.2). The definition of the locale `coset` resides in the theory `Coset` that is built *via* theory extension on the theory of groups. Thereby, we can use the locale `group`

[1]The definition of left cosets depicted here uses the image operation ``. We use the same for right cosets, deviating from the formalization seen in Sylow's proof. The definitions are equivalent.

and on top of it the newly defined locale coset and perform proofs about cosets in an intelligible and concise way. The theorems we need for Sylow's proof mentioned in Section 3.2.4 become more readable. For example, the antecedent of coset_mul_assoc has been without locales

```
r_coset G  (r_coset G M g) h = r_coset G M (bin_op G g h)
```

In the scope of the locale coset we can write this equation as

```
(M #> g) #> h = M #> (g # h)
```

The advantage is considerable, especially if we consider that the syntax is not only important when we type in the goal for the first time, but we are confronted with it in each proof step. Hence, the syntactical improvements are crucial for a good interaction with the proof assistant.

The theorems we derive about the set product of groups are needed as a calculational basis for the theorems involving the factorization of a group (see Section 6.3.1). The binary operation of groups is lifted to the level of subsets of a group. We derive algebraic rules relating the coset operators <# and #> with the product operation for subsets <#>. A couple of examples giving a gist of the derived rules and of the syntactical improvements we gain by locales are

```
[| H <| G; H1 ∈ {* H *}; x ∈ H1 |] ==> I(H1) = H #> (i x)

[| H <| G; x ∈ (G.<cr>); y ∈ (G.<cr>) |]
  ==> (H #> x) <#> (H #> y) = H #> (x # y)

[| H <| G; H1 ∈ {* H *}; H2 ∈ {* H *} |] ==> H1 <#> H2 ∈ {* H *}
```

These preparations give us the appropriate frame for further experiments into abstract algebraic proof. At the same time, the basic theorems entail sufficient algebra for Sylow's theorem. Before we get into further examples, we shortly reconsider the proof of Sylow's theorem and the improvements we achieved by the new concepts.

6.2 Sylow's Theorem

In the case study of Sylow, we were forced to abuse the theory mechanism to achieve readable syntax for the main proof. We declared the local constants and definitions as Isabelle constants and definitions. To model local rules, we used axioms, *i.e.* Isabelle rules. This works well, but contradicts the meaning of axioms and definitions in a theory (*cf.* Section 2.1.1).

Locales offer the ideal support for this procedure and it is methodically sound. In the theory of cosets, we define a locale for the proof of Sylow's first theorem.

The natural number constants we had to define before as constants of an Isabelle theory become now locale constants. The names we use as abbreviations for larger formulas like the set $\mathcal{M} \equiv \{S \subseteq G_{cr} \mid order(S) = p^{\alpha}\}$ also become added as locale constants. So, the `fixes` section of the locale `sylow` is

```
locale sylow = coset +
  fixes
    p  :: "nat"
    a  :: "nat"
    m  :: "nat"
    calM      :: "'a set set"
    RelM      ::  "('a set * 'a set)set"
```

The following `defines` section introduces the local definitions of the set $\mathcal{M}$ and the relation $\sim$ on $\mathcal{M}$ (here `calM` and `RelM`).

```
defines
  calM_def "calM == {s. s ⊆ (G.<cr>) & card(s) = (p ^ a)}"
  RelM_def "RelM == {(N1,N2).(N1,N2) ∈ calM × calM
                   & (∃ g ∈ (G.<cr>). N1 = (N2 #> g) )}"
```

Note that the previous definitions depend on the locale constants `p`, `a`, and `m` (and `G` from locale `group`). The example illustrates the advantage we gain from locales. We can abbreviate in a convenient way using locale constants without being forced to parameterize the definitions, *i.e.* without locales we would have to write `calM G p a m` and `RelM G p a m`. Furthermore, without locales the definitions of `calM` and `RelM` would have to be theory level definitions, whereas now they are just local.

Finally, we add the locale assumptions to the locale `sylow`. Here, we can state all assumption that are local for the 52 theorems of the Sylow proof. In the mechanization of the proof without locales in Chapter 3 all these merely local assumptions had to become rules of the theory for Sylow.

```
assumes
  Group_G    "G ∈ Group"
  prime_p    "p ∈ prime"
  card_G     "order(G) = (p ^ a) * m"
  finite_G   "finite (G.<cr>)"
```

The locale `sylow` can subsequently be opened to provide just the right context to conduct the proof of Sylow's theorem in the way we discovered in Chapter 3 to be appropriate. But in contrast to there, the assumptions and definitions have only local significance. This makes the formalization clear and readable.

Apart from the already obvious advantages we gain from locales, there are two more important improvements illustrated by the recapitulation of the Sylow case study which we want to emphasize by a separate consideration in the subsequent sections.

6.2.1 Adapted Polymorphism

As already mentioned on the conceptual level in Chapter 4, the declarations of locale constants may use polymorphism, but this is different to the usual one. The use of the same type variable name for the declaration of different constructors in an Isabelle theory does not imply any connection between definitions using the same variable. In locales, we enrich the expressivity of polymorphic definitions by extending the scope of the polymorphic variable names over all constant declarations of a locale (*cf.* Section 4.3.3). This changes the usual polymorphism of Isabelle, in that equal names imply the same variable. That is, polymorphic variables are fixed by the variable names, *e.g.* `'a`, inside the locale. Although the locale as an entity can still be instantiated to arbitrary constant types of appropriate sort, the instantiation is implicitly forced to be the same for all constants of a locale that use the same variable name.

This restriction only holds if the same names are used. Naturally we preserve the same freedom of expressivity that was there before: if we use different variable names in polymorphic declarations of locale constants, they can be instantiated independently. An example for this will be given in Section 6.3.2.

However, in the Sylow case study — and in abstract algebraic applications in general — this is exactly what we need: we want to constrain different constructors to the same type, while we still want to stay abstract, *i.e.* use polymorphic declarations. In the *ad hoc* mechanization of Sylow's theorem in Chapter 3 we had to declare a fixed type `i` to achieve the same effect that we gain now from the locale concept. There, we had to use the fixed type `i` to be able to assume the (local) properties for the proof of Sylow's theorem. The assumption of these properties for a polymorphic constant `G::'a set` would have made the formalization unsound.

Obviously, the trick used for Sylow's theorem in Chapter 3 is not at all satisfying because the result is not applicable to groups of arbitrary type; although `i` is a random type, it is fixed. This is a severe restriction and proves that this particular feature of locales is not only convenient but also necessary. Furthermore, it illustrates that locales are valuable in their own right in that they implement a more explicit control for polymorphism.

6.2.2 More Encapsulation

In the *ad hoc* approach to the mechanization of Sylow's theorem, we had no means to encapsulate local assumptions and definitions. Although we abused the theory mechanism to that end, we did not structure as far as it seemed possible, because it felt wrong to use theory level definitions and rules to an extent that would make the entire proof look dubious.

Now, we may soundly use the locale mechanism for any merely locally

relevant definition. In particular we can define the abbreviation

```
H == {g. g ∈ (G.<cr>) & M1 #> g = M1}
```

for the main object of concern, the Sylow subgroup that is constructed in the proof. Naturally, we refrained from using a definition for this set before because in the global theorem it is not visible at all, *i.e.* it is a temporary definition. But, by adding the above line to a new locale, after introducing a suitably typed locale constant in the `fixes` part,

```
H :: "'a set"
```

the proofs for Sylow's theorem improve a lot. A further measure taken now is to define in the new locale the two assumptions that are visible in most of the 52 theorems in the file for Sylow's proof. Summarizing, a locale for the central part of Sylow's proof is given by:

```
locale sylow_central = sylow +
  fixes
    H  :: "'a set"
    M  :: "'a set set"
    M1 :: "'a set"
  assumes
    M_ass  "M ∈ calM / RelM &
           ¬(p ^ ((max-n r. p ^ r | m)+ 1) | card(M))"
    M1_ass "M1 ∈ M"
  defines
    H_def  "H == {g. g ∈ (G.<cr>) & M1 #> g = M1}"
```

We open this locale after the first few lemmas when we arrive at theorems that use the locale assumptions and definitions. Subsequently, we assume that the locales `group` and `coset` are open. Henceforth, the conjectures become shorter and more readable than in the *ad hoc* version. For example,

```
[| M ∈ calM / RelM
   & ¬(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
   M1 ∈ M; x ∈ {g. g ∈ (G.<cr>) & M1 #> g = M1};
   xa ∈ {g. g ∈ (G.<cr>) & M1 #> g = M1} |]
   ==> x # xa ∈ {g. g ∈ (G.<cr>) & M1 #> g = M1}
```

can now be stated as

```
[| x ∈ H; xa ∈ H |] ==> x # xa ∈ H
```

The shorthand for `H` is particularly handy when one has to instantiate schematic variables of theorems explicitly using, for example, the Isabelle tactic `res_inst_tac`.

Of all the lemmas that are proved inside the scope of the innermost locale `sylow_central`, we just export the theorems

```
∃ M1. M1 ∈ M                                                    [existsM1inM]
∃ M. M ∈ calM / RelM &                                            [lemmaA1]
  ¬(p ^ ((max-n r. p ^ r | m)+ 1) | card(M))
H <<= G & card(H) = p ^ a                                       [main_proof]
```

For the export of these three theorems we use the one step export function `Export` that only normalizes the theorems up one level (*cf.* Section 4.3.2). Thereby, they become normalized according to the scope of the locale `sylow`. If we apply `Close_locale "sylow_central"` to pop `sylow_central` off the scope, we reestablish a scope on which we have `group`, `coset`, and `sylow` open. This is just the right context to prove the final theorem `Sylow1`

```
∃ H. H <<= G & card(H) = p ^ a
```

The proof is easily derived from `main_proof` by using existential elimination and the other two previously exported theorems that are just of the right form with respect to the locale scope.

If we finally export this theorem using the generally normalizing function `export`, we achieve the desired form of Sylow's theorem which is completely independent from any local definitions and assumptions made so far. The theorem stands alone as a global theorem of the Isabelle theory `Coset`, and is hence applicable to any group. This becomes visible in the resulting theorem by the question marks indicating schematic variables.

```
[| ?p ∈ prime; finite (carrier ?G); ?G ∈ Group;
   order ?G = ?p ^ ?a * ?m |] ==> ∃ H. H <<= ?G & card H = ?p ^ ?a
```

6.3 More Abstract Algebra

In this section we present some more examples from abstract algebra and lattice theory that we mechanized in Isabelle. They further validate the concepts put forth in this thesis. In particular, they illustrate the need for dependent types in addition to locales to express structural properties sufficiently well.

6.3.1 Factorization of a Group

If a group is factorized by one of its normal subgroups then the factorization together with the induced operations on the cosets is again a group. This is a quite standard result of group theory, but it is challenging because it contains a self-reference: a structure constructed from a group shall be a group again. We define the factorization of a group using the concept of structures as dependent types of Chapter 5. In addition, we will analyze to what extent locales can be helpful.

For this proof we define a new theory that builds on the theory of cosets. The factorization of a group by one of its normal subgroups is given by

the set of cosets. The operations on the cosets are described by the group operations lifted to the level of cosets, *i.e.* the binary operation is given by the product of cosets, the inverse operation is given by the inverse coset, and the factor of the factorization serves as a unit element, a normal subgroup H. To describe this construction formally, we use the typed λ-notation from Chapter 5.

```
FactGroup ==
  λ G ∈ Group. λ H ∈ Normal ↓ G.
  (| carrier = set_r_cos G H,
     bin_op = λ X ∈ set_r_cos G H.
              λ Y ∈ set_r_cos G H. set_prod G X Y,
     inverse = λ X ∈ set_r_cos G H. set_inv G X,
     unit = H |)
```

The relation between the input parameters and the image of a higher order structure for group factorization is functional. The factorization is a unique construction because we can explicitly identify how it is built by the coset operations. That is why we can define `FactGroup` as an element of a higher order structure using λ.

We define the theory syntax `G Mod H` for the factorization `FactGroup G H`. To enhance the readability of the construction and thereby the proofs about it, we employ locales. We cannot use any nicer syntax in the above definition of the factorization because in the body of the λ-term above, the terms `G` and `H` are parameters. Hence, they have to stay flexible. But, using locales we can fix a group `G` and a normal subgroup `H` in `G` for the local proof context.

```
locale factgroup = coset +
  fixes
    F :: "('a set) grouptype"
    H :: "('a set)"
  assumes
    H_ass "H <| G"
  defines
    F_def "F == FactGroup G H"
```

By defining this locale as an extension of the locale `coset`, we incorporate all the syntactical abbreviations we defined for cosets and operations on cosets in Section 6.1.2. In addition, we have the group `G` already as a fixed local constant. The additional definition of the factorization as `F` lets us derive in the scope of this locale[2]

```
F = (| carrier = {* H *},
       bin_op = (λ X ∈ {* H *}. λ Y ∈ {* H *}. X <#> Y),
       inverse = (λ X ∈ {* H *}. I(X)),
       unit = H |)
```

[2]Opening `factgroup` automatically opens `coset` and `group`.

The derivation is an application of Isabelle's simplifier to the corresponding definitions, and the reduction rules for λ. By the additional use of the locale properties of fixing and local definition, we achieve a readable syntax in a local scope.

With these preparations, we can prove that this factorization is again a group, which is trivially stated as `F ∈ Group` in the scope of the locale. The proof is straightforward. By backward resolution with the introduction rule for the group property `GroupI` we can reduce it to six subgoals that can be solved by repeatedly applying previously derived results about cosets and the operations on them. Note that here the initial application of `GroupI` illustrates the first class property of locales we gain through the export, as discussed in Section 4.8. Although we proved the rule `GroupI` for the fixed group `G` we can now apply it again to the group `F` which is constructed with that same `G`.

By exporting the result that `F` is a group we get the general formula

```
[| ?G ∈ Group; ?H <| ?G |]                          [FactorGroup_is_Group]
  ==> ?G Mod ?H ∈ Group
```

In an earlier version of this experiment, we did not employ locales. The statement of the conjecture was even without locales not such a problem; it corresponded to the above formula. But, in the proof of the group property, where all the definitions of the lifted operations have to be employed, we were formerly exposed to formulas that were hard to read. For example, the sixth subgoal of the proof corresponding to the proof of the associativity property of the group is after one simplification step that reduces the λ-terms without locales

```
[| G ∈ Group; H <| G; x ∈ set_r_cos G H; y ∈ set_r_cos G H;
       z ∈ set_r_cos G H |]
==> set_prod G (set_prod G x y) z = set_prod G x (set_prod G y z)
```

The use of locales improves this to

```
[| x ∈ {* H *}; y ∈ {* H *}; z ∈ {* H *} |]
    ==> x <#> y <#> z = x <#> (y <#> z)
```

As a further illustration of the concept of higher order structures, we consider the proof that the constructed λ-term `FactGroup` is an element of a suitable Π-set.

```
FactGroup ∈ (Π G ∈ Group. (Normal ↓ G) -> Group)
```

This membership statement is equivalent to the structural proposition that the factorization of a group is a function mapping a group and a normal subgroup of this group to another group. We call this kind of theorem a *structural proposition* because membership in a structure (a set) entails the proposition we just proved, *i.e.* that the factorization is a group. If

we interpret the sets that are our structures as types, then we see how the Curry-Howard isomorphism of propositions-as-types is embodied in a statement like above (*cf.* Section 5.3.2). In contrast to type theory, we do not need to state this isomorphism as a paradigm — it is inherent because we use sets: from the above we can derive the logical proposition.

6.3.2 Direct Product of Groups

The direct product of two groups G and G' is a structure built over the set of pairs of elements (x, y) where $x \in G$ and $y \in G'$. This set, together with the pointwise product of the operations of the groups G and G', is again a group. This theorem is, like the previous example, a test for the adequacy of our approach. We have elements of a structure that are combined together and build again an element of that same structure, a self-referential aspect that can only be mechanized with first class structures.

Similar to the previous example, we define the direct product of two groups at the theory level using an element of Π.

```
ProdGroup ==
λ G ∈ Group. λ G' ∈ Group.
 ⦇ carrier = (G.<cr>)×(G'.<cr>),
   bin_op = (λ (x, x') ∈ (G.<cr>)×(G'.<cr>).
             λ (y, y') ∈ (G.<cr>)×(G'.<cr>).
                  ((G.<f>) x y, (G'.<f>) x' y')),
   inverse = (λ (x, y) ∈ (G.<cr>)×(G'.<cr>).
                ((G.<inv>) x, (G'.<inv>) y)),
   unit = ((G.<e>),(G'.<e>)) ⦈
```

We define the theory syntax ⦇ G1, G2 ⦈ for this direct product of two groups.

To enhance the proof process we can employ again locales to reduce the complex definition of `ProdGroup` to a readable formula. Otherwise the above term will occur in the proof of the group property.

For the second group `G'` we define a locale `r_group`

```
locale r_group = group +
  fixes
    G'        :: "'b grouptype"
    e'        :: "'b"
    binop'    :: "'b => 'b => 'b"      ("(_ #' _)" [80,81]80 )
    INV'      :: "'b => 'b"            ("i' (_)"      [90]91)
  assumes
    Group_G' "G' ∈ Group"
  defines
    e'_def     "e' == (G'.<e>)"
    binop'_def "x #' y == (G'.<f>) x y"
    inv'_def   "i'(x) == (G'.<inv>) x"
```

The locale r_group is basically a repetition of the locale group. As in the case of the factorization in the previous section, we build a new syntactic layer into the locales for this example.

```
locale prodgroup = r_group +
  fixes
    P :: "('a * 'b) grouptype"
  defines
    P_def "P == ⦇ G, G' ⦈"
```

We separated the two locales because the former one is independently useful for other examples reasoning about two groups.

First, we can simplify the definition of ProdGroup by considering the term that represents the product in the scope of the locale prodgroup because G and G' are fixed there.

```
P =
⦇ carrier = P.<cr>,
   bin_op = (λ (x, x') ∈ (P.<cr>). λ (y, y') ∈ (P.<cr>).
                          ( x # y, x' #' y')),
   inverse = (λ (x, y) ∈ (P.<cr>). (i x, i' y)),
   unit = P.<e> ⦈
```

The additional syntactical layer that is created by the locale prodgroup enables to stay syntactically abstract during the proof of the group property of the direct product. That is, if we prove that the direct product is a group

```
P ∈ Group
```

we have a clear syntax. In the proof we can use most of the time the abbreviation P and only unfold when it is necessary. Mostly, we are not confronted with the gory details of the construction — the simplifier is able to solve this task when we add the locale definitions to the simplification sets. The exported result is independent of the locales and applicable to any two groups.

```
[| G ∈ Group; G' ∈ Group |] ==> ⦇ G, G' ⦈ ∈ Group
```

As in the factorization example, we consider the membership of the direct product in a higher order structure — in this case again a Π-type.

```
ProdGroup ∈ Group -> Group -> Group
```

As with the factorization of groups, we first performed the mechanization without the use of locales[3]. In comparison, we could reduce the size of the proof by 50% using locales. Although in the latter version some savings are due to polishing the proofs by improving the applications of automatic simplification tactics, a larger portion is due to locales. Furthermore, the

[3]At the time the implementation was not capable of dealing with nested locales.

streamlining of the proofs was made much easier because of the greatly improved comprehensibility. Where we were lost before in grasping huge complicated terms, and thus sometimes misled from the optimum solution, the natural representation achieved by locales leads the way now.

6.3.3 Group of Bijections

To prepare some further results concerning groups of automorphisms we prove a general result about maps: the set of bijections with the appropriate operations of composition of bijections, inverse bijection, and identical bijection form a group.

We define the set of bijections `Bij` and a record `BijGroup` made up out of the set of bijections over a set S, the composition of these bijections, the inverse of a bijection and the identical bijection.

```
Bij S == {f | f ∈ S -> S & f(S) = S & inj_on S f}
BijGroup S == (| carrier = Bij S,
                 bin_op = λ f ∈ Bij S. λ g ∈ Bij S. g ∘S f,
                 inverse = λ f ∈ Bij S. λ x ∈ S. (Inv S f),
                 unit = λ x ∈ S. x |)
```

The constructors $\circ_S$ and `Inv S` are the composition and the inverse of the typed λ-functions we defined for the Π-sets (see Chapter 5). We can show that this record is in the set `Group`, *i.e.* that the bijections together with the listed operations on them are a group

```
BijGroup S ∈ Group                                    [Bij_are_Group]
```

The method used is the same as for the previous two examples: we define a locale to provide concise notation.

6.3.4 Group of Ring Automorphisms

With this preparation we can attempt a proof concerning a special class of homomorphisms: automorphisms. They are bijective homomorphisms from an algebraic structure to itself. The example described in this section is the proof that the automorphisms of a ring form a group.

Definitions

Rings are defined in a similar manner to groups. The definition of rings is not relevant for the present example. It will be described in Section 6.4.1 where its nested structure is of interest. Assuming the definition of group homomorphisms Hom from Section 5.2.2, group automorphisms can now be defined as homomorphisms from a group to itself that are bijective on the carrier of the group. Ring automorphisms are defined in a very similar way.

```
constdefs
  GroupAuto :: "('a grouptype * ('a => 'a)) set"
  "GroupAuto == Σ G ∈ Group. {Φ. (G,G,Φ) ∈ Hom  &
                 inj_on Φ (G.<cr>) & Φ `` (G.<cr>) = (G.<cr>)}"

  RingAuto :: "(('a ringtype) * ('a => 'a))set"
  "RingAuto == Σ R ∈ Ring. {Φ. (R,R,Φ) ∈ RingHom &
                 inj_on Φ (R.<cr>) & Φ `` (R.<cr>) = (R.<cr>)}"
```

Proof

Using the definition of ring automorphisms `RingAuto`, we show that the set of ring automorphisms is a subgroup of the group of bijections over the carrier of the ring. Since we know that any subgroup is itself a group (`subgroupE2`) we get the group property of ring automorphism straightforwardly. But, the proof that they are a subgroup is much simpler than showing that ring automorphisms are a group explicitly. This is evident if we look at the obligations we have to show for the subgroup property. They are entailed as premises of the subgroup introduction rule we derived as one of the classical theorems about subgroups (see Section 6.1.1). This rule says that it is sufficient to show that a subset H of a group G is closed under the group operations in order to infer that H is a subgroup of G.

```
[| H ⊆ carrier G; H ≠ {}; ∀ a ∈ H. i(a) ∈ H;                [subgroupI]
   ∀ a ∈ H. ∀ b ∈ H. a # b ∈ H |] ==> H <<= G
```

To show the group property explicitly we would need to show the six group properties as encoded in the rule `GroupI`. But now, if we prove the theorem

```
R ∈ Ring ==>                                          [RingAuto_SG_Bij_Group]
RingAuto ↓ R  <<= BijGroup (R.<cr>)
```

we can use the result that the set `BijGroup` is a group shown in the previous section. By applying prederived basic rules for subgroups we obtain immediately from the former theorem that the ring automorphisms together with the appropriate operations are a group.

```
R ∈ Ring ==>                                            [RingAuto_are_Group]
(| carrier = RingAuto ↓ R,
   bin_op = λ x ∈ RingAuto ↓ R.
            λ y ∈ RingAuto ↓ R. (BijGroup(R.<cr>).<f>) x y,
   inverse = λ x ∈ RingAuto ↓ R. (BijGroup(R.<cr>).<inv>) x
   unit = BijGroup(R.<R>).<e> |) ∈ Group
```

The Isabelle proof code that produces this result is short. It just concatenates the former theorems using forward resolution `RS`.

```
RingAuto_SG_BijGroup RS (Bij_are_Group RS subgroupE2);
```

It illustrates nicely how the first class representation of structures allows the reduction of the proposition and hence improves the proof process.

6.3.5 Tarski

The fixed point theorem of Tarski [Tar55] is well known in computer science. Yet the form of the theorem which is usually proved is an older version from 1928. This theorem says that the least upper bound of all fixed points of a monotonic function f over a complete lattice $(A, \sqsubseteq)$ can be obtained as $\bigvee\{x \in A \mid x \sqsubseteq f(x)\}$. The dual is true for the greatest lower bound $\bigwedge$. Besides proving that, Tarski showed in the later paper that the set of all fixed points of f is itself a complete lattice. This second result is very well suited to illustrate the need for a proper structural representation, because it is proved by applying the first part of the theorem to the interval sublattice $[\bigvee Y, 1]$ for any subset Y of the set of all fixed points. So, our mechanized proof illustrates again the advantages of the present approach.

The proof of this full version was first formalized by R. Pollack in LEGO [Pol90]. There, partial equivalence relations have to be used, which make the proof quite hard to read. That is, in LEGO we are deprived of the natural notion of equality to be able to express the self-referential aspects of this theorem.

Our formalization is, like the other examples, based on an adequate representation of lattices as structures and uses locales in addition to improve the syntax and shorten the formulas and proofs.

6.4 Operations on Modules

Through the embedding of structures as Σ-types and Π-types, we achieve first class representations of modules. Thereby, we are able to use structures in formulas. Hence, we can express general operations on structures such as unions of forgetful functors. We illustrate this in the present section.

A ring can be considered as a construction from an abelian group and a semigroup over the same carrier. The definition we used — and that is used in mathematical textbooks — is the *ad hoc* definition of a ring. To illustrate the adequacy of our concepts we show how the substructure of a ring that is an abelian group can be revealed and how conversely a ring can be seen as constituted by a group and a semigroup. The former is provided by a typed function that belongs to a class of functions that are known as forgetful functors.

6.4.1 Forgetful Functor

For the rather simple forgetful functor example we first have to explain how we formalized rings. Using extension of record types, we can build the base type for rings on the base type for groups `grouptype`.

```
record 'a ringtype = 'a grouptype +
  Rmult    :: "['a, 'a] => 'a"
```

Thereby, we inherit the components of groups and can form rings by just extending the latter by the second operation `Rmult`[4]. We add the syntax `R.<m>` for the additional element `Rmult` of a ring to adapt the notation for rings to the syntax of the group projections.

To isolate the group contained in a ring we can use an element of a Π-set. This λ-function represents a forgetful functor. It "forgets" some structure.

```
constdefs
  group_of :: "'a ringtype => 'a grouptype"
   "group_of == λ R ∈ Ring.
       ⦇ carrier = (R.<cr>), bin_op = (R.<f>),
         inverse = (R.<inv>), unit = (R.<e>) ⦈"
```

Thereby, we are able to refer to the substructure of the ring that forms an abelian group using the forgetful functor `group_of`. We can derive the theorem[5]

```
R ∈ Ring ==> group_of R ∈ AbelianGroup                    [R_Abel]
```

This should enable a better structuring and decomposition of proofs. In particular, we can use this functor when we employ locales for ring related proofs. Then we want to use the encapsulation already provided for groups by the locale `group`. To achieve this we define the locale for rings as an extension.

```
locale ring = group +
  fixes
    R     :: "'a ringtype"
    rmult :: "['a, 'a] => 'a" (infixr "**" 80)
  assumes
    Ring_R "R ∈ Ring"
  defines
    rmult_def "x ** y == (R.<m>) x y"
    R_id_G    "G == group_of R"
```

Note that we are able to use the locale constant `G` again in a locale definition. This is sound because we have not defined `G` yet. If one gives a constant an inconsistent definition, then one will be unable to instantiate results proved in the locale. This way of reusing the local proof context of groups for the superstructure of rings illustrates the flexibility of locales as well as the ease of integration with the mechanization of structures given by Σ and Π.

Naturally, one could have used a different setup for rings in which the entire structure would have been defined separately from groups. The advantage would have been to avoid the close connection with the functor

[4]Note that we formalize rings without 1. Mathematical textbooks sometimes use the notion of rings for rings with a 1 for convenience.

[5]`AbelianGroup` is the structure of abelian, *i.e.* commutative, groups. Their definition is a simple extension from the one of groups by the additional commutativity.

group_of. The disadvantage would have been to abandon the perfectly suited locale context for groups.

There is a slight drawback with this intricate use of locales: theorems that are proved in the setup of the locale rings using group results will have in the exported form the assumptions

```
[| R ∈ Ring; group_of R ∈ Group ... |] ==> ...
```

But, as an implication of the theorem R_Abel, we can easily derive

```
R ∈ Ring ==> group_of R ∈ Group
```

Thus, the second premise can be canceled. Although we have to do a final proof step to cancel the additional premise, this shows the advantage of locales being dissolved into premises of global representations of theorems: it is impossible to introduce unsoundness. A definition of a locale constant that is not consistent with its properties stated by locale rules would be not applicable. Since locale assumptions and definitions are explained through meta-assumptions, the resulting theorem would carry the inconsistent assumptions implicitly. The final proof step we do in the example seems a small price we have to pay for a smooth merging of locales and structures.

6.4.2 Union: Construction of a Ring

A ring may be constructed from an abelian group and a semigroup where this semigroup shares the same carrier with the group.

We refine the definition of groups to entail semigroups. To that end, we enclose the carrier and the binary operation in a record definition for semigroups.

```
record 'a semigrouptype =
  carrier  :: "'a set"
  bin_op   :: "['a, 'a] => 'a"
```

We then build the definition of the type for groups as an extension of the type semigroup. The definition of the structure of semigroups is just a subformula of the group definition (see Section 5.2.2).

To build a ring, the binary operations of an abelian group and a semigroup have to obey the right and left distributive laws

```
distr_l S f1 f2 == ∀ x ∈ S. ∀ y ∈ S. ∀ z ∈ S.
                     f1 x (f2 y z) = f2 (f1 x y) (f1 x z)
distr_r S f1 f2 == ∀ x ∈ S. ∀ y ∈ S. ∀ z ∈ S.
                     f1 (f2 y z) x = f2 (f1 y x) (f1 z x)
```

assuming that f1 is the semigroup operation, f2 the group operation, and S the carrier of the group.

For the actual construction of the ring, we define a typed λ-function ring_from.

```
ring_from :: "['a grouptype, 'a semigrouptype] => 'a ringtype"
```

If we consider the class of abelian groups **AbelianGroup** and a structure of semigroups that shares the group's carrier and fulfills the distributivity laws we can define this λ-function as

```
λ G ∈ AbelianGroup.
   λ S ∈ {M. M ∈ Semigroup & (M.<cr>) = (G.<cr>)
           & distr_l (G.<cr>) (M.<f>) (G.<f>)
           & distr_r (G.<cr>) (M.<f>) (G.<f>)}.
       (| carrier = (G.<cr>), bin_op = (G.<f>),
          inverse = (G.<inv>), unit = (G.<e>), Rmult = (S.<f>) |)
```

That is, we construct an element of the appropriate record type for rings from elements of the two structures **AbelianGroup** and a specialized substructure of **Semigroup**. We can derive

```
ring_from G S ∈ Ring
```

if `G` and `S` are elements of the parameter structures of **ring_from**. More generally, we can derive a structural proposition that entails the logic of the construction of the ring.

```
ring_from ∈
(Π G ∈ AbelianGroup.
   { M. M ∈ Semigroup & (M.<cr>) = (G.<cr>)
    & distr_l (G.<cr>) (M.<f>) (G.<f>)
    & distr_r (G.<cr>) (M.<f>) (G.<f>)}   -> Ring)
```

The examples for constructions and deconstructions of rings in this section are classical operations of theories, *i.e.* the first one is a forgetful functor and the second one a union of two structures with sharing and constraints on parameters (the distributivity of the operations).

Summarizing, we have illustrated in this chapter three things. First, we have shown how the use of locales improves the proving process. Second, we have illustrated that we can represent algebraic structures adequately by the mechanization of Π and Σ. And finally, the presented case studies illustrate that the two concepts can smoothly be combined to provide the same support as modules but — through the separation of the concepts of locality and structures — more adequately.

Chapter 7

Conclusions

In this final chapter, we summarize in Section 7.1 the work we performed and highlight major decisions. Subsequently, we list the improvements of our concepts, evaluating to what extent the objectives were met. Section 7.2 names the major achievements. In Section 7.3, we discuss the experiences and the lessons we learned from performing the project. Finally, we end this thesis with a general concluding remark in Section 7.4.

7.1 Summary

This work is concerned with modules for theorem provers, in particular Isabelle. Modules provide a means for structuring theories and representing mathematical structures. The latter mechanism is typical for applications in abstract algebra. We set out in Chapter 1 with the hypothesis that modules should have a representation in the logic, *i.e.* be first class citizens. According to our hypothesis an amalgamation of aspects of locality provided by modules and adequacy through the first class property of structures is the solution for higher expressivity, good performance, and intelligible representation of abstract algebra.

We began the research by looking at related work in Chapter 2. There we compared module systems of interactive theorem provers and their applicability to abstract algebra. Our comparison of theorem proving systems comparable to Isabelle validated our hypothesis that their modules are not adequate for abstract algebraic reasoning. Furthermore, we investigated a different family of proof systems based on type theory in Section 2.4. They use an interesting alternative to modules: some powerful type theories use dependent types to represent structures. These modules are first class citizens.

Next we checked the hypothesis by doing a large case study: a mechanical proof of Sylow's theorem in Chapter 3. The evaluation of this case study shows on the one hand that locality is crucial for algebraic proofs and on

the other hand that the first class structure representation is a necessity for adequate reasoning.

Drawing from the experience of this case study in group theory, we develop a concept of *locales* (see Chapter 4) that captures local definitions, pretty printing syntax, and local assumptions. Locales provide support for locality by realizing local contexts for Isabelle proofs. They can be seen as a simple form of modules.

According to our initial hypothesis, we naturally explored the possibility to find, and generate automatically, a first class representation for locales. In a prototypical experiment, we identified objects that are representations of locales at the meta-level (see Section 4.7). Those are meta-level predicates. Since Isabelle's meta-logic is only a fragment of higher order logic, we would need to extend the meta-logic of Isabelle with explicit conjunction and (dependent) products to be able to encode structures adequately as meta-level predicates. But, this extension is not always desirable. The need for an explicit first class representation only partly applies to locales. The aspect of locality is often used *without* a need to identify a first class notion for a local proof context. That is, a general concept of locales that applies to all object logics turns out to be useful as a stand-alone feature. Examples for such local proof contexts are the locales `sylow` and `sylow_central` in Section 6.2. These locales enclose a local context for a certain proof but do not represent any algebraic or other structure. We have to distinguish between an algebraic structure and a local context for a proof. Therefore, it is sensible to separate concerns and keep locales as a separate general concept for locality.

However, locales are not sufficient to describe structuring adequately. For example, structures like groups and rings need an explicit representation as objects in the logic. A mechanization of dependent Σ-types and Π-types as typed sets in higher order logic is presented in Chapter 5. The mechanization of Σ-types and Π-types as HOL sets is necessary because there are no dependent types in HOL. The dependent types can represent structures adequately in the logic because these structures are then terms, *i.e.* first class citizens.

The combination of locales with dependent types provides a sufficient methodology to formalize abstract algebra in a higher order logic theorem prover: in Chapter 6 we validate the combination of the two concepts on case studies.

7.2 Achievements

There are three main achievements in this work. Firstly, we mechanized Sylow's theorem. This case study is an interesting formalization experiment in its own right [KP99]. Secondly, we designed and implemented locales. This

concept is implemented and released with Isabelle version 98-1 [KW98]. As a concept it is valuable beyond the scope of Isabelle. Finally, we mechanized a first class representation of modular structures [Kam99] in such a way that it can easily be combined with the locale concept. Thereby, we realized modular reasoning in Isabelle.

Locales are a sectioning concept for higher order logic theorem provers.

- The concept of locales proves to enhance the reasoning with abstract algebraic proofs dramatically. This is illustrated by the Sylow case study. The requirements set out by the *ad hoc* version of Sylow in Chapter 3 are fulfilled by application of locales (see Section 6.2).

- Locales are a general concept of local proof contexts. The concept is general: it can be transferred to any higher order logic theorem prover (see Section 4.8).

- Locales realize a form of polymorphism with binding of type variables not normally possible in Isabelle (see Section 6.2.1).

The concept of structures as dependent types addresses the adequate representation of module systems.

- The Σ and Π-sets enable first class representations for modular structures. Thereby, adequate representations of algebraic structures become possible. Several non-trivial case studies could be performed and illustrate the capability of the concept.

- The concepts are relatively light-weight as they are based on simple formalizations. At the same time the Π and Σ-types are strong enough to express higher-level modular notions, like mappings between parameterized structures. We can represent abstract operations on structures like forgetful functors and unions of structures with sharing.

Most prominently, we achieved the goals we set out with by applying our concepts of modular reasoning in combination.

- Structures expressed by Σ and Π-sets can be complicated. By using locales in addition they can be greatly simplified inside the scope of a locale. This enhances comprehension of the formulas and consequently facilitates the proving process. Examples for this have been presented in Section 6.3.

- The encapsulation provided by locales enhances proofs and shortens their presentation. If proofs are concerned with structures that build on other structures the corresponding locales can model this structural extension by the locale extension device (see Section 4.3.2).

- To be able to decompose higher-level structures so that they can actually make efficient use of underlying locale theorems, maps between structures can be defined. They can be employed to make reuse of locales possible. This is illustrated by the example of rings built on groups and the interaction with the forgetful functor `group_of` in Section 6.4.1.

7.3 Lessons Learned

Initially, we aimed at constructing one module system that would have *both* features: locality and adequacy. We have already named various reasons and gave evidence that support the decision to separate these major concerns.

There are some other interesting experiences we made during design, prototyping, and implementation that support the separation from a pragmatic point of view:

- The implementation of locales was very simple once the right idea was found to realize local constants with dependent pretty printing syntax.
- The mechanization of dependent types as sets is similarly simple and light-weight. Through derived introduction and elimination rules the structures can be reduced in an application almost as easily[1] as the built in λ-expressions of Isabelle/HOL.
- The combination of the concepts is trivially possible, *i.e.* there is no other device necessary to use them in combination.

During the construction and experimentation phase of the concepts we naturally encountered difficulties. We found it quite difficult to integrate the prototype for locales into Isabelle. Fortunately, the concept for theory data (see Section 4.4) provided a systematic way for the integration. However, this concept is very abstract. Hence, the integration of locales into Isabelle was harder than the construction of the initial prototype. Without a systematic concept it is likely that the integration would have turned into a very complicated task, though.

Naturally, our work raised some questions that can only be answered through further experimentation.

- Isabelle uses simplification sets for rewriting. We decided not to put locale definitions into (local) simplification sets because we do not want these definitions to be unfolded automatically. However, there are situations when we need to unfold locale definitions. More experience with the use of locales might show if there is a systematic way of supporting such cases.

[1] Although the terms are simply reduced, there may be nontrivial membership obligations produced.

- Dependent types are represented as sets in our mechanization. Some of the proofs we performed using dependent types correspond to a type-inference for dependent types. This is undecidable in general and a tactic trying to solve it could fail. Nevertheless, similar to the mechanism generating type checking condition in PVS, one might try to automatize this to some extent. Although Isabelle's general tactics support the proof of such obligations quite well, experiments towards implementing a specific tactic that implements a type-inference algorithm for our dependent sets would be desirable.

- The examples with structures as dependent types we considered in Section 6.4 show that with the tool set of Σ and Π we can define operations on modules, like unions or forgetful functors. Although our structures are values, we cannot express such operations generally because we did not construct an explicit notion of signature and structure. There is some theoretical work on theories for theorem provers [LB92]. The speciality of this approach is that theories are values, and there is an explicit notion of signatures, axioms, and theories. It defines a general notion of so-called *frames* that generalizes the notions of theories for theorem proving systems. The work develops a lattice of signatures that enables to construct general operations on signatures and structures using joins and meets. Since our structures are values, it might be possible to construct the lattice of signatures for structures as dependent types in Isabelle. Thereby, we would gain more generality in construction as well as reasoning with structures.

On the whole, locales, dependent types and their combination show the advantage of using elegant ideas from type theory without sacrificing pragmatic advantages of the LCF approach.

7.4 Concluding Remark

To conclude this work, we would like to return to our argument from the first few pages. One intention of the thesis was to show that formalizations of abstract algebra are a useful test bench for the development of concepts for theorem provers. The formal proof of Sylow's theorem in Isabelle did have a positive impact on the development of techniques for interactive generic theorem proving in higher order logics. The concepts we developed were based on experience from abstract algebra. The use is by no means restricted to abstract algebra: locales can be used throughout Isabelle's object logics and are already used to conduct proofs about the formal method UNITY (see Section 4.8). Apart from the conceptual improvements, the work may thus be seen as a contribution to strengthen the reputation of formalizations of mathematics.

Appendix A

Theorems for Sylow's Proof

This appendix serves as a lookup section for Chapter 3. Selected theorems of group theory, some that deal with the combinatorial argument, and all from the theory for Sylow's theorem are contained here. We do not display any proofs. We left the syntax as ASCII in order to give a real impression of the Isabelle code.

The prefix ! is the universal quantifier, : represents $\in$ for HOL sets, and the question mark '?' is the existential quantifier of HOL. The symbol % stands for λ; Un for set union $\cup$, ~ for $\neg$, and <= for $\subseteq$ or $\leq$.

A.1 Group Theory

```
coset_mul_assoc      "[| G : Group; M <= (G.<cr>); g : (G.<cr>);
                     h : (G.<cr>) |] ==> r_coset G (r_coset G M g) h
                     = r_coset G M ((G.<f>) g h)";

coset_mul_unity      "[| G: Group; x : (G.<cr>); H <<= G; x : H |]
                     ==> r_coset G H x = H";

coset_mul_invers1    "[| G: Group; x : (G.<cr>); y : (G.<cr>);
                     M <= (G.<cr>);
                     r_coset G M ((G.<f>) x ((G.<inv>) y)) = M |]
                     ==> r_coset G M x = r_coset G M y";

coset_mul_invers2    "[| G: Group; x : (G.<cr>); y : (G.<cr>);
                     M <= (G.<cr>); r_coset G M x = r_coset G M y|]
                     ==> r_coset G M ((G.<f>) x ((G.<inv>) y)) = M ";

coset_join1          "[| G: Group; x : (G.<cr>); H <<= G;
                     r_coset G H x = H |] ==> x : H";

coset_join2          "[| G: Group; x : (G.<cr>); H <<= G;
                     x : H |] ==> r_coset G H x = H";
```

```
set_r_cos_part_G      "[| G: Group; H <<= G|]
                      ==> Union (set_r_cos G H) = (G.<cr>)";

rcosetGHa_subset_G    "[| G: Group; H <= (G.<cr>); a : (G.<cr>) |]
                      ==> r_coset G H a <= (G.<cr>)";

card_cosets_equal     "[|G : Group; H <= (G.<cr>);
                      finite((G.<cr>))|] ==>
                      ! c: set_r_cos G H. card c = card H & finite c";

r_coset_disjunct      "[| G: Group; H <<= G |] ==>
                      ! c1: set_r_cos G H. ! c2: set_r_cos G H.
                      c1 ~= c2 --> c1 Int c2 = {}";

set_r_cos_subset_PowG "[| G: Group; H <<= G |]
                      ==> set_r_cos G H <= Pow( (G.<cr>))";

Lagrange              "[| G: Group; finite(G.<cr>); H <<= G |]
                      ==> card(set_r_cos G H) * card(H) = order(G)";
```

A.2 Combinatorics

```
n_choose_0            "(n choose 0) = 1";

zero_le_choose        "k <= n ==> 0 < (n choose k)";

less_choose           "n < k ==> (n choose k) = 0";

n_choose_n            "(n choose n) = 1";

choose_Suc            "(Suc n choose n) = Suc n";

n_choose_1            "n choose 1 = n";

div_order             "[| k | n; 0 < n |] ==> k <= n";

max_p_div             "[| 1 < p; 0 < s |] ==> p ^ log p s | s &
                      (! m. log p s < m --> ~(p ^ m | s))";

unique_max_power_div_s
                      "[| 1 < p; 0 < s |] ==>
                      (max-n r. p ^ r | s) = log p s";

log_p_unique          "[| 1 < p; 0 < s |] ==> ?! x.  p ^ x | s &
                      (! m. x < m --> ~(p ^ m | s))";

max_p_div_eq_log      "[| 1 < p; 0 < s;
                      p ^ x dvd s & (! m. x < m  --> ~(p ^ m dvd s))|]
                      ==>  log p s = x";
```

```
div_eq_log_p          "[|1 < p; 0 < a ; 0 < b;
                      ! (r :: nat). ((p ^ r | a) = (p ^ r | b))|]
                      ==>  log p a = log p b";

log_power_div_equality
                      "[|1 < p; 0 < n |] ==>
                      n = (p ^ log p n)*(n div (p ^ log p n))";

equiv_partition       "[| finite S; equiv S rel; ! x : S / rel.
                      k dvd card(x)& finite x |] ==> k dvd card(S)";

constr_bij            "[| finite M; x ~: M |] ==>
                      card {s. ? s1 :
                      {s. s <= M & card(s) = k}. s = insert x s1}
                      = card {s. s <= M & card(s) = k}";

n_subsets             "[| finite M; card(M) = n; k <= n |] ==>
                      card {s. s <= M & card(s) = k} = (n choose k)";

Rettung               "[| 0 < m; 0 < k;
                      k < (p^a); (p^r) | (p^a)* m - k |]
                      ==> r <= a";

p_fac_forw            "[| 0 < m; 0 < k; p : prime;
                      k < (p^a); (p^r) | (p^a)* m - k |]
                      ==> (p^r) | (p^a)- k";

r_le_a_forw           "[| 0 < k; k < (p^a);
                      0 < p; (p^r) | (p^a) - k |] ==> r <= a";

p_fac_backw           "[| 0 < m; 0 < k; p : prime;
                      k < (p^a); (p^r) | (p^a)- k |]
                      ==> (p^r) | (p^a)* m - k";

logp_eq_logp          "[| p : prime; 0 < m |] ==>
                      ! k n. (k < (p ^ a) & n < (p ^ a) * m &
                      0 < k & 0 < n & n - k = (p ^ a) * m -  (p ^ a) &
                      (p ^ a) <= (p ^ a) * m & k <= n)
                      --> log p k = log p n";

p_not_div_choose      "[| p : prime; ! k n. (k < p1 & n < p2 & 0 < k &
                      0 < n & n - k = p2 - p1 & p1 <= p2 & k <= n)
                      --> log p k = log p n ;p1 <= p2|] ==>
                      k <= n & k < p1 & n < p2 & n - k = p2 - p1
                      --> ~(p | (n choose k))";
```

```
const_p_fac_right   "[| p : prime; 0 < m |] ==>
                    ~(p | ((p ^ a) * m - 1 choose (p ^ a) - 1))";

const_p_fac         "[| p : prime; 0 < m |] ==>
                    (max-n r :: nat. (p ^ r | (m :: nat))) =
                    (max-n r.
                    (p ^ r | (((p ^ a) * m) choose p ^ a)))";
```

A.3 Theory Sylow

```
RelM_equiv          "equiv calM RelM";

M_subset_calM       "M : calM / RelM &
                    ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M))
                    ==> M <= calM";

card_M1             "[|M : calM / RelM &
                    ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                    M1 : M|] ==> card(M1) = p ^ a";

exists_x_in_M1      "[|M : calM / RelM &
                    ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                    M1 : M|] ==>  ? x. x : M1";

M1_subset_G         "[| M : calM / RelM &
                    ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                    M1 : M|] ==>   M1 <= (G.<cr>)";

M1_inj_H            "[| M : calM / RelM &
                    ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                    M1 : M|] ==>
                    ? f: {g. g : (G.<cr>) & M1 #> g = M1} -> M1.
                    inj_on f {g. g : (G.<cr>) & M1 #> g = M1}";

RangeNotEmpty       "[| {} = RelM ^^ {x}; x : calM |] ==> False";

EmptyNotInEquivSet  "{} ~: calM / RelM";

existsM1inM         "M : calM / RelM &
                    ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M))
                    ==> ? M1. M1 : M";

zero_less_o_G       "0 < order(G)";

zero_less_m         "0 < m";

card_calM           "card(calM) = ((p ^ a) * m choose p ^ a)";

max_p_div_calM      "~(p ^ ((max-n r. p ^ r | m)+ 1) | card(calM))";
```

```
finite_calM          "finite calM";

lemma_A1             "? M. M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M))";

bin_op_closed_lemma  "[| M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                     M1 : M; x : {g. g : (G.<cr>) & M1 #> g = M1};
                     xa : {g. g : (G.<cr>) & M1 #> g = M1}|]
                     ==> x # xa : {g. g : (G.<cr>) & M1 #> g = M1}";

H_is_SG              "[|M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                     M1 : M |] ==>
                     {g. g : (G.<cr>) & M1 #> g = M1} <<= G";

M_elem_map           "[| M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                     M1: M; M2: M|] ==> ? g: (G.<cr>). M1 #> g = M2";

H_elem_map           "[|M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M)); M1:M;
                     H : set_r_cos G {g. g : (G.<cr>) & M1 #> g = M1}
                     |] ==> ? g: (G.<cr>).
                     {g. g : (G.<cr>) & M1 #> g = M1} #> g = H";

rcosetGM1g_subset_G  "[| M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                     M1: M; g : (G.<cr>); x : M1 #> g |]
                     ==> x : (G.<cr>)";

finite_M1            "[|M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                     M1: M|] ==> finite M1";

finite_rcosetGM1g    "[|M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                     M1: M; g : (G.<cr>)|] ==> finite (M1 #> g)";

M1_cardeq_rcosetGM1g "[| M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                     M1: M; g : (G.<cr>)|]
                     ==> card(M1) = card(M1 #> g)";

M1_RelM_rcosetGM1g   "[| M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                     M1: M; g : (G.<cr>)|] ==> (M1, M1 #> g) : RelM";
```

```
bij_M_GmodH          "[| M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                     M1 : M|] ==>
                     (? f: M ->
                     set_r_cos G {g. g : (G.<cr>) & M1 #> g = M1}.
                     inj_on f M ) &
                     (? g: (set_r_cos G {g. g: (G.<cr>) & M1#>g=M1})
                      -> M. inj_on g
                     (set_r_cos G {g. g: (G.<cr>) & M1 #> g = M1}))";

calM_subset_PowG     "calM <= Pow((G.<cr>))";

finite_M             "M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M))
                     ==> finite M";

cardMeqIndexH        "[| M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                     M1 : M |] ==>
                     card(M) =  card(set_r_cos G
                     {g. g : (G.<cr>) & M1 #> g = M1})";

index_lem            "[| M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                     M1 : M|] ==>
                     (card(M)*card({g. g : (G.<cr>) & M1 #> g = M1}))
                     = order(G)";

lemma_leq1           "[| M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                     M1 : M |] ==>
                     p ^ a <=
                     card({g. g : (G.<cr>) & M1 #> g = M1})";

lemma_leq2           "[| M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                     M1 : M |] ==>
                     card({g. g : (G.<cr>) & M1 #> g = M1})
                     <= p ^ a";

main_proof           "[| M : calM / RelM &
                     ~(p ^ ((max-n r. p ^ r | m)+ 1) | card(M));
                     M1 : M|] ==>
                     {g. g : (G.<cr>) & M1 #> g = M1} <<= G &
                     card({g. g : (G.<cr>) & M1 #> g = M1}) = p ^ a";

Sylow1               "? H. H <<= G & card(H) = p ^ a";
```

Bibliography

[Asp91] D. R. Aspinall. Isabelle Modules – A New Theory Mechanism for Isabelle. Master's thesis, University of Cambridge, 1991.

[Bai98] A. Bailey. *The Machine-Checked Literate Formalisation of Algebra in Type Theory.* PhD thesis, University of Manchester, 1998.

[Bur90] R. Burstall. Computer Assisted Proof for Mathematics: an Introduction using the LEGO Proof System. Technical Report ECS-LFCS-91-132, University of Edinburgh, 1990.

[C+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[Chu40] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, pages 56–68, 1940.

[CPM90] T. Coquand and C. Paulin-Mohring. Inductively Defined Types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[D+93] Gilles Dowek et al. The Coq proof assistant user's guide. Technical Report 154, INRIA-Rocquencourt, 1993.

[dB80] N. G. de Bruijn. A Survey of the Project AUTOMATH. In Seldin and Hindley [SH80], pages 579–606.

[dB91] N. G. de Bruijn. Telescoping Mappings in Typed Lambda Calculus. *Information and Computation*, 91:189–204, 91.

[Dow90] G. Dowek. Naming and Scoping in a Mathematical Vernacular. Technical Report 1283, INRIA, Rocquencourt, 1990.

[Far90] W. M. Farmer. A Partial Functions Version of Church's Simple Theory of Types. *Journal of Symbolic Logic*, 55:1269–91, 1990.

[Far93] W. M. Farmer. A Simple Type Theory with Partial Functions and Subtypes. *Annals of Pure and Applied Logic*, 64:211–240, 1993.

[FFL97] S. Finn, M. P. Fourman, and J. Longley. Partial Functions in a Total Setting. *Journal of Automated Reasoning*, 18:85–104, 1997.

[FGT92a] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: System Description. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 701–705. Springer-Verlag, 1992.

[FGT92b] W. M. Farmer, J. D. Guttman, and F. J. Thayer. Little Theories. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.

[FGT93] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: an Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.

[FGT95] W. M. Farmer, J. D. Guttman, and F. J. Thayer. The IMPS User's Manual – First Edition, Version 2. Technical report, The MITRE Corporation, 1995.

[FGT98] W. Farmer, J. Guttman, and F. J. Thayer. IMPS Theory Library Home Page. available from the web: file//math.harvard.edu/imps/imps_html/theory-library.html, 1998.

[Fle99] J. D. Fleuriot. *A Combination of Geometry and Nonstandard Analysis, with Application to Newton's Principia.* PhD thesis, Computer Laboratory, University of Cambridge, 1999. forthcoming.

[GGH90] Stephen J. Garland, John V. Guttag, and James J. Horning. Debugging Larch Shared Language specifications. *IEEE Transactions on Software Engineering*, 16(9):1044–1075, September 1990. Reprinted as DEC Systems Research Center Report 60. Superseded by Chapter 7 in [GH93].

[GH91] John V. Guttag and James J. Horning. A tutorial on Larch and LCL, a Larch/C interface language. In S. Prehn and W. J. Toetenel, editors, *VDM91: Formal Software Development Methods*, Delft, October 1991. Springer-Verlag Lecture Notes in Computer Science 551. Superseded by Chapter 3 of [GH93].

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[GHM90] John V. Guttag, James J. Horning, and Andrés Modet. Report on the Larch Shared Language: Version 2.3. Report 58, DEC Systems Research Center, Palo Alto, CA, April 14 1990. Superseded by Chapter 4 of [GH93].

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL, a Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[Gor95] M. J. C. Gordon. Merging HOL with Set Theory. Technical report, University of Cambridge, Computer Laboratory, 1995.

[Gor96] M. J. C. Gordon. Set Theory, Higher Order Logic or Both? In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *LNCS*, pages 191–202. Springer-Verlag, 1996. Revised version.

[GP93] M. J. C. Gordon and A. M. Pitts. The HOL Logic. In Gordon and Melham [GM93], pages 191–232.

[Gun89] E. L. Gunter. Doing Algebra in Simple Type Theory. Technical Report MS-CIS-89-38, Dep. of Computer and Information Science, University of Pennsylvania, 1989.

[Gun90] E. L. Gunter. The Implementation and Use of Abstract Theories in HOL. In *Third HOL Users Meeting*, Aarhus University, 1990.

[Hal60] Paul R. Halmos. *Naive Set Theory.* Van Nostrand, 1960.

[Har95] J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. Available on the Web as `http://www.cl.cam.ac.uk/users/jrh/papers/reflect.dvi.gz`.

[Her64] I. N. Herstein. *Topics in Algebra.* Xerox, 1964.

[Hey56] A. Heyting. *Intuitionism: An Introduction.* North Holland, Amsterdam, 1956.

[Hic96] J. J. Hickey. Formal Objects in Type Theory Using Very Dependent Types. In *Foundations of Object Oriented Languages 3*, 1996. Available on the Web as `http://www.cs.cornell.edu/jyh/papers/fool3.ps`.

[Hic97] J. J. Hickey. Nuprl-Light: An Implementation Framework for Higher-Order Logics. In *International Conference on Automated Deduction, CADE-14*, volume 1249 of *LNCS*. Springer-Verlag, 1997.

[Hol] The HOL System, Tutorial. Available on the Web as `http://lal.cs.byu.edu/lal/holdoc/tutorial.html`.

[How80] W.A. Howard. The formulae-as-types notion of construction. In Seldin and Hindley [SH80], pages 479–490.

[Jac95] P. B. Jackson. *Enhancing the NUPRL Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, Department of Computer Science, 1995.

[JM93] B. Jacobs and T. F. Melham. Translating Dependent Type Theory into Higher Order Logic. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in LNCS, pages 209–229. Springer-Verlag, 1993. Utrecht, March 16 - 18.

[Kam99] F. Kammüller. Modular Structures as Dependent Types in Isabelle. In *TYPES '98*, volume 1657 of *LNCS*. Springer-Verlag, 1999. Selected papers. To appear.

[KP99] F. Kammüller and L. C. Paulson. A Formal Proof of Sylow's First Theorem – An Experiment in Abstract Algebra with Isabelle HOL. *Journal of Automated Reasoning*, 1999. To appear.

[KPS+92] S. Kromodimoeljo, B. Pase, M. Saaltink, D. Craigen, and I. Meisels. The Eves System. In *International Lecture Series on Functional Programming, Concurrency, Simulation and Automated Reasoning(FPCSAR)*, McMaster University, August 1992.

[KW98] F. Kammüller and M. Wenzel. Locales – a Sectioning Concept for Isabelle. Technical Report 449, University of Cambridge, Computer Laboratory, 1998.

[LB92] Z. Luo and R. Burstall. A Set-theoretic Setting for Structuring Theories in Proof Development. Technical Report ECS-LFCS-92-206, Department of Computer Science, 1992.

[LP92] Z. Luo and R. Pollack. Lego proof development system: User's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

[LP97] L. Lamport and L. C. Paulson. Should Your Specification Language Be Typed? (revised version). Technical report, 1997. Original version available as Report 425, Computer Lab (1997).

[Luo90a] Z. Luo. A Higher-order Calculus and Theory Abstraction. *Information and Computation*, 90(1), 1990.

[Luo90b] Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990. Also as Report CST-65-90.

[Luo92] Z. Luo. A Unifying Theory of Dependent Types: the Schematic Approach. In *Symposium on Logical Foundations of Computer Science*, volume 620 of *LNCS*. Springer-Verlag, 1992.

[Mac86] D. B. MacQueen. Using Dependant Types to Express Modular Structures. In *Proc. 13th ACM Symp. Principles Programming Languages.* ACM Press, 1986.

[Mel93] T. F. Melham. The HOL Logic Extended with Quantification over Type Variables. *Formal Methods in System Design*, 3(1-2):7–24, 1993.

[MS95] S. P. Miller and M. Srivas. Formal verification of the aamp5 microprocessor – a case study in the industrial use of formal methods. Technical report, 1995. presented at WIFT '95.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, 1991.

[Nip93] T. Nipkow. Axiomatic Type Classes (in Isabelle). In *Types for Proofs and Programs*, Nijmegen, 1993.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory — An Introduction.* Oxford Science Publications. Clarendon Press, Oxford, 1990.

[NW98] W. Naraschewski and M. Wenzel. Object-oriented Verification based on Record Subtyping in Higher-Order Logic. In *11th International Conference on Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*, ANU, Canberra, Australia, 1998. Springer-Verlag.

[OSRSC98] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. PVS Language Reference. Part of the PVS Manual. Available on the Web as `http://www.csl.sri.com/pvsweb/manuals.html`, September 1998.

[Pau89] L. C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[Pau90] L. C. Paulson. Isabelle: The Next 700 Theorem Provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Pau91a] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[Pau91b] L. C. Paulson. Theories as ML Structures, Signatures, and Functors. Unpublished, University of Cambridge, 28th January 1991. Third Draft.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[Pau95] L. C. Paulson. First Isabelle User's Workshop. Technical Report 379, Computer Laboratory, University of Cambridge, September 1995.

[Pau98] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6:85–128, 1998.

[PG96] L. C. Paulson and K. Grabczewski. Mechanizing Set Theory. *Journal of Automated Reasoning*, 17:291–323, 1996.

[Pol90] R. Pollack. The Tarski Fixpoint Theorem. e-mail to: proof-sci@se.chalmers.cs, 1990.

[QED96] QED. The "hohum" objection, April 1996. Mailing list discussion.

[Rus92] J. M. Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 237–257, Nijmegen, The Netherlands, January 1992. Springer Verlag.

[Saa89] M. Saaltink. A Formal Description of Verdi. Technical Report 89-5429-10, ORA Canada, October 1989.

[SB83] D. T. Sannella and R. M. Burstall. Structured Theories in LCF. In *CAAP'83: Trees in Algebra and Programming*, volume 159 of LNCS, pages 377–91. Springer-Verlag, 1983.

[SC91] M. Saaltink and D. Craigen. Simple Type Theory in EVES. In G. Birtwistle, editor, *4th Workshop on Higher Order Logic*, 1991. Springer Verlag.

[SH80] J.P. Seldin and J.R. Hindley, editors. *To H. B. Curry: Essays on Combinatory Logic*, Academic Press Limited, 1980.

[Sin97] S. Singh. *Fermat's Last Theorem.* Fourth Estate, 1997.

[Sli98] K. Slind. HOL98 Draft User's Manual. Available on the Web at: `http://www.cl.cam.ac.uk/users/kxs/`, 1998.

[SM95] M. K. Srivas and S. P. Miller. Formal Verification of an Avioniocs Microprocessor. Technical Report SRI-CSL-95-4, SRI and Collins Commercial Avionics, 1995.

[Syl72] Ludwig Sylow. Théorémes sur les groupes de substitutions. *Mathematische Annalen*, 5:584–594, 1872.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[Try93] A. Trybulec. Some Features of the Mizar Language. 1993. Available from Mizar user's group.

[Typ98] Workshop Types. Working Group Meeting. Kloster Irsee, March 1998.

[Wen95] M. Wenzel. Using Axiomatic Type Classes in Isabelle. Draft, August 1995.

[Wie59] H. Wielandt. Ein Beweis für die Existenz der Sylowgruppen. *Archiv der Mathematik*, 10:401–402, 1959.

[Wil95] A. Wiles. Modular elliptic curves and Fermat's Last Theorem. *Annals of Mathematics*, 142:443–551, 1995.

[Win93] P. J. Windley. Abstract Theories in HOL. In L. Claesen and M. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, IFIP Transactions A-20, pages 197–210. North-Holland, 1993.

[WR62] A. N. Whitehead and B. Russell. *Principia Mathematica.* Cambridge University Press, 1962. Paperback edition to *56, abridged from the 2nd edition (1927).

[Yu90] Y. Yu. Computer Proofs in Group Theory. *Journal of Automated Reasoning*, 6:251–286, 1990.