Isabelle/Isar — a versatile environment for human-readable formal proof documents

Markus M. Wenzel

Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

Institut für Informatik
der Technischen Universität München
Lehrstuhl für Software & Systems Engineering

Isabelle/Isar — a versatile environment for human-readable formal proof documents

Markus Michael Wenzel

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Christoph Zenger

Prüfer der Dissertation:

1. Univ.-Prof. Tobias Nipkow, Ph.D.
2. Univ.-Prof. Dr. Helmut Schwichtenberg
   Ludwig-Maximilians-Universität München

Die Dissertation wurde am 25. September 2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 30. Januar 2002 angenommen.

Zusammenfassung

Diese Arbeit möchte maschinelle Beweise einem breiteren Publikum zugänglich machen. Die hierzu eingeführte formale Sprache *Isar* erlaubt *Beweis Dokumente* auf einem für menschliche Leser angemessenen Niveau zu verfassen. Die logische Fundierung erfolgt durch Interpretation als abstrakte Inferenzen im Isabelle System. Die Isabelle/Isar Umgebung ist generisch bezüglich Objekt-Logiken und Beweiswerkzeugen, und unterstützt gleichermaßen Natürliches Schließen sowie algebraische Umformungen. Anwendungen aus der Logik, Mathematik und Informatik belegen die Vielseitigkeit und Praxistauglichkeit der Isar Konzepte.

Abstract

The basic motivation of this work is to make formal theory developments with machine-checked proofs accessible to a broader audience. Our particular approach is centered around the *Isar* formal proof language that is intended to support adequate composition of *proof documents* that are suitable for human consumption. Such *primary proofs* written in Isar may be both checked by the machine and read by human-beings; final presentation merely involves trivial pretty printing of the sources. Sound logical foundations of Isar are achieved by interpretation within the generic Natural Deduction framework of Isabelle, reducing all high-level reasoning steps to primitive inferences.

The resulting Isabelle/Isar system is generic with respect to object-logics and proof tools, just as pure Isabelle itself. The full Isar language emerges from a small core by means of several derived elements, which may be combined freely with existing ones. This results in a very rich space of expressions of formal reasoning, supporting many viable proof techniques. The general paradigms of Natural Deduction and Calculational Reasoning are both covered particularly well. Concrete examples from logic, mathematics, and computer-science demonstrate that the Isar concepts are indeed sufficiently versatile to cover a broad range of applications.

Acknowledgements

I am indebted to numerous people who have influenced this work in one way or the other (in alphabetical order): Andreas Abel, David Aspinall, Gertrud Bauer, Henk Barendregt, Stefan Berghofer, Bernd Grobauer, John Harrison, Florian Kammüller, Gerwin Klein, Ralph Matthes, Stephan Merz, Olaf Müller, Wolfgang Naraschewski, Tobias Nipkow, David von Oheimb, Larry Paulson, Leonor Prensa Nieto, Cornelia Pusch, Norbert Schirmer, Helmut Schwichtenberg, Monika Seisenberger, Sebastian Skalberg, Konrad Slind, Martin Strecker, Freek Wiedijk, and Vincent Zammit.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The general idea of "formalizing" human reasoning already has a long tradition, reaching at least back to ancient Greek philosophy. The "calculemus" manifest of Leibniz may be seen as a more recent incident, aimed to supersede philosophical disputes by a formalized process to "decide" the truth of statements. Purely syntactic underpinning of formal reasoning (with mechanical checking of proofs) has finally matured during the 20th century, although the claims on universal truth had to be dismissed. Roughly speaking in the first half of the century, logicians have demonstrated that mathematics could in principle be completely reduced to a few logical principles. In the second half of the century, the advent of computers has enabled to build systems for actually doing non-trivial developments in formal logic.

At some point in the history of computer-based reasoning, visionaries of "artificial intelligence" proclaimed that it was possible to build *fully automated* theorem provers that would be able to conduct substantial mathematical proof developments without human intervention! Whereas many automated reasoning techniques have been devised over several decades, they have proven to be rather limited in practice, being useful only for restricted technical problems.

Over the last 10–20 years, a different tradition of *interactive theorem proving* has become quite successful in supporting reasonably sized formal theory developments. Interactive proving proceeds by instructing the machine (the "proof checker") step-by-step, until the intended result is achieved eventually. The individual reasoning steps may vary in granularity, ranging from single rules to invocations of automated proof procedures (for local problems). Such *semi-automated reasoning* systems have been able to cover significant applications from areas of pure logic, mathematics and computer-science (e.g. mathematical background theories, abstract models of hardware and software systems, programming language semantics, algorithms and functional programs). Note that

new (deep) mathematical results are normally *not* discovered by proof development on the machine. There are also some practical limits on verifying concrete hardware or software systems at large (this market has basically moved over to model-checking and testing lately, trading actual verification for systematic finding of errors).

Despite the relative success of theorem proving in certain areas, there are still fundamental obstacles in addressing a broader range of users, even those with some interest in formal logic and proof itself. The full potential of applications of semi-automated reasoning has probably not been unleashed yet.

The rather paradoxical problem is that most interactive proof systems do not support an adequate notion of proof (speaking from the human perspective). Major systems are still too much oriented towards technical issues of certain logical calculi and their implementation on the machine. The input is given by slightly arcane languages of *proof scripts*, which are difficult to understand later on; proof developers typically need to replay existing scripts to recover some idea of the actual reasoning process. This situation is bad enough for proof maintenance, but is impossible for communicating formal proofs to a wider audience. It also poses a particular problem for derivative work, based on the formal theory development of previous authors.

From the methodological viewpoint, interactive proof development is similar to programming, although proving is slightly more involved in practice. Successful proof checking typically demands a good portion of experimentation, either to convince the machine of "obviously correct" reasoning steps, or to figure out "minor omissions" in the initial claim (or the underlying definitions). On the other hand, theorem proving has the fundamental advantage that the intended results (i.e. theorems) are usually completely specified in advance, so the subsequent proof may only fail, but not produce a wrong result (at least for sound implementations of proof checkers). In principle, this enables proof scripts to be expressed in an arbitrarily "bad" manner without affecting the result.

From the perspective of programming language design, which has undergone several decades of research itself, proof development still seems to be stuck at the assembly language level. There have been several attempts to improve the technological backdrop of theorem proving, again by drawing from common ideas of program development. Notable approaches include various user interfaces for theorem provers, presentation and management of sources (e.g. as in "literate programming"), visualization or verbalization of machine-oriented proof structures (e.g. by natural language generation), specific infrastructure for "proving in the large" (via module systems, change management of proofs etc.), and large-scale repositories of theory developments (e.g. "mathematical knowledge bases"). All of these are legitimate research issues within the vicinity of formal proof development, but they do not address the core problem of low-level proof representations in the first place.

We shall illustrate this discrepancy again in terms of programming. The effort required to work with a slightly low-level language like "C" may be consider-

ably reduced by external tool support, e.g. by automatic code generation from abstract graphical presentations that are composed by means of a nice user-interface. Such an environment merely uses "C" as a fronted, but does not overcome its inherent deficiencies, which become again relevant if the output needs to be augmented manually. In contrast, a considerably more powerful programming language (like ML or Haskell) could have provided first-class presentations of high-level concepts itself, eliminating the need for heavy tool support in the first place. (Some kind of tool support could still become useful at a later stage.)

Our work is motivated by the perceived lack of accessible proof representations for semi-automated reasoning. The Isabelle/Isar environment to be introduced here is intended as a viable basis for human-readable proof documents, which are composed by the user and checked by the machine. The main focus will be on the theory and practice of the *Isar proof language* ("Isar" abbreviates "Intelligible semi-automated reasoning"). We particularly aim at preserving important factors that have made interactive theorem proving a success so far. We also intend to cover at least the same range of applications as existing interactive provers, but expect to unleash further potential of semi-automated reasoning due to the new quality of formal proofs achieved by Isar.

1.2 Related work

Over the last few decades, a large number of systems have been built that are intended for "theorem proving" in one way or the other. The overview of "mathematics in the computer" [Wiedijk, 2001a] lists over 130 entries in the categories of "First Order Prover", "Logic Education", "Proof Checker", "Tactic Prover", or "Theorem Prover". This list only covers systems that are sufficiently significant and still available.

In order to point out relevant related work of Isabelle/Isar we shall briefly review a few notable systems, both "real" working environments and some recent studies on human-readable proof representations. Further discussions of existing approaches will be given later on, alongside of our exposition of Isar itself.

1.2.1 Real theorem proving environments

Theorem proving systems that qualify as "real" working environments are typically based on expressive formal languages (type theory or set theory), feature user-guided proof development (usually interactive), and have been successfully applied "in reality" by a considerable number of users. Below we consider important representatives of this category: HOL, Coq, Isabelle, PVS, and Mizar. There certainly exist a few further contenders in the same league, such as Nuprl [Constable *et al.*, 1986] or ACL2 [Kaufmann *et al.*, 2000].

HOL

There is a whole family of HOL systems, which all share the same logical foundations and system architecture. The "official" line is represented by HOL88 [Gordon, 1988] [Gordon and Melham, 1993], HOL90 (K. Slind), and the still current HOL98 (K. Slind and M. Norrish). There have been a few side branches as well, including the commercial implementation ProofPower (R. Arthan of ICL Secure Systems), and HOL-Light [Harrison, 1996a] which has been successfully employed for industrial verification tasks of floating point arithmetic. See also [Gordon, 2000] for further background information on HOL and its relatives.

The HOL logic is based on a version of Church's "Simple Theory of Types" [Church, 1940] [Henkin, 1950] [Andrews, 1986], which has been extended by schematic polymorphism, first-order type constructors, and a semantic type definition scheme [Gordon, 1985a] [Gordon, 1985b] [Pitts, 1993]. The HOL methodology emphasizes a strictly definitional discipline of theory development; arbitrary axiomatizations are largely considered harmful by the user community. Starting from the rather small axiomatic basis of primitive HOL, standard mathematical concepts may be developed with reasonable effort. Over the years, HOL users have collected a large body of material.

The system architecture of HOL follows the pioneering approach of LCF [Gordon *et al.*, 1979], based on Milner's "Correctness by Construction" principle. Here a small trusted kernel implements primitive inferences of the basic logic, using a strongly-typed functional programming language such as ML. Any further functions written by users may never "invent" new theorems, but are restricted to the abstract theorem constructors of the kernel (by virtue of type-safety of the programming language). Thus one achieves a high degree of reliability, while avoiding to store actual proof objects for independent checking (which is required for systems implemented in untyped languages like LISP, such as AUTOMATH [de Bruijn, 1980] [Nederpelt *et al.*, 1994]). The "LCF architecture" has enabled efficient implementation of many advanced proof tools (e.g. rewriting, classical proof procedures) and derived specification mechanisms (such as inductive sets and types, cf. the discussion in [Harrison, 1995]), without affecting soundness of the logical core. Even more importantly, contributors need not understand the underlying logic in full detail.

HOL does not enforce a standard paradigm to produce proven results. In principle, primitive and derived rules (written in ML) may be invoked directly, mapping existing theorems to new ones. Nevertheless, most users follow the goal-oriented view of tactical theorem proving: an initial claim is refined by backwards steps until a solved form is achieved. The ML specification for such transformation steps is quite hard to follow in general, even if the original writer has refrained from any ad-hoc programming, restricting the script to standard tactics and tactic combinators. HOL is often perceived as rather cryptic for outsiders, not only due to the inherent complexities of tactical proof scripts, but also due to the details of concrete syntax within raw ML. There is also no clear distinction between extending and using the system, as both involves ML.

There have been several attempts to organize HOL proofs in a more accessible manner, e.g. by generating textual reports on the dynamic evolution of goal states [Cohn, 1995]. Alternative proof styles (still within the tactical approach) have been proposed as well, e.g. a generalized version of calculational reasoning called "window inference" [Grundy, 1991]. Some notable experiments on structured proof languages within HOL have been conducted as well, see §1.2.2.

Coq

Coq [Barras *et al.*, 1999] essentially draws from the same tradition of interactive theorem proving as the HOL family, but follows a rather different philosophy in many respects.

The logical foundation of Coq is the "Calculus of Inductive Constructions" (CIC), i.e. a constructive type theory with builtin notions of inductive types and recursive functions [Pfenning and Paulin-Mohring, 1990] [Paulin-Mohring, 1993]. Proofs are internally represented as dependently typed λ-terms, which are explicitly stored for separate checking by a distinctive system component. Even though Coq has been implemented in a type-safe programming language, Milner's "Correctness by Construction" of LCF/HOL has been given up in favor of the venerable "de-Bruijn principle", with independent checking of static proof objects [de Bruijn, 1980] (see also the survey of [Barendregt and Geuvers, 2001]). In practice, both Coq and HOL achieve a similar level of reliability, but Coq demands significantly more time and space resources in realistic applications.

Coq provides particular infrastructure to extract functional programs from constructive proofs. In principle, the internal λ-term structure of proof objects may be automatically compiled to produce ML code. In practice, users interested in program extraction need to be careful to conduct proofs properly, in order to arrive at programs conforming to their intention. In particular, concepts need to be arranged appropriately at the level of inductive *Set* or logical *Prop* types.

Coq renounces the free programmability of HOL, but offers separate languages for theory specifications (called "Gallina") and tactical proof scripts, respectively. Here the raw ML view has been successfully replaced by sane concrete syntax. Users may still implement their own proof tools, but this is rarely required in practice. Coq generally provides much less automated proof support than HOL: whereas existing classical first-order techniques of automated reasoning may be used within classical higher-order logic quite easily, proof search within a constructive setting is much more involved. Interestingly, many Coq users who are interested in large applications tend to introduce non-constructive axioms in the very beginning in order to ease the formalization effort (even though this breaks the program extraction facility). This constructively incorrect tuning of the formal basis would in principle admit more powerful proof procedures, but official Coq does not support classical reasoning specifically.

Coq tactic scripts and primitive proof terms are both largely inaccessible to human readers. Traditionally, some of the key developers of Coq have been

more interested in getting formal proofs accepted by the machine at all (and maybe extract programs later), rather than achieve nice presentations for human readers. Nonetheless, significant work on rendering primitive proof objects (λ-terms) in natural language (English or French) has been undertaken in the past [Coscoy *et al.*, 1995]. A similar verbalization facility is provided by the Minlog system [Benl *et al.*, 1998] (for its own proof terms). The HELM project [Asperti *et al.*, 2001] aims at WWW access of formal theories at large (currently working mainly for Coq), but the fundamental problems of adequate representation of proof terms are still there, despite the XML document view provided here.

Isabelle

Isabelle [Paulson and Nipkow, 1994] is positioned as a "generic theorem proving environment" according to the LCF/HOL tradition of interactive systems, but is aimed to support many logics. According to its original author the early history of Isabelle is a "tale of errors, not grand designs" [Paulson, 1990]. Apart from the generic framework (Isabelle/Pure), the Isabelle distribution includes concrete object-logics that are ready for immediate applications, notably Isabelle/HOL [Nipkow *et al.*, 2001] [Nipkow and Paulson, 2001] (simply-typed classical set-theory), Isabelle/HOLCF [Regensburger, 1995] [Müller *et al.*, 1999] (domain theory within HOL), and Isabelle/ZF [Paulson, 1993] [Paulson, 1995] (untyped set-theory according to Zermelo-Fraenkel).

The Isabelle/Pure framework implements minimal higher-order logic, with unrestricted universal quantification "$\bigwedge$", implication "$\Longrightarrow$", and equality "$\equiv$". Rules formulated via $\bigwedge/\Longrightarrow$ may be composed by higher-order resolution (which also involves higher-order unification) [Paulson, 1986]. Resolution is the most fundamental reasoning principle of Isabelle, it admits both forward and backward chaining of natural-deduction rules [Paulson, 1989]. Derived rules are represented directly as meta-level theorems, eliminating the need for hand-written ML code as in the LCF/HOL family. Generic higher-order rewriting (by means of $\equiv$ rules) is also available, as well as classical reasoning tools [Paulson, 1997] [Paulson, 1999] that may be instantiated for many important logics.

Formalizing new object-logics (following natural-deduction principles) is quite easy in Isabelle, merely by providing a few declarations of abstract and concrete syntax, and primitive proof rules. On the other hand, a realistic working environment like Isabelle/HOL demands many years of further work in order to develop a sufficiently rich library of standard mathematical concepts. Practical applications also demand advanced specification mechanisms (which need to be implemented separately), notably inductive sets and types [Paulson, 1994] [Berghofer and Wenzel, 1999], and recursive functions [Slind, 1996] [Slind, 1997].

The majority of Isabelle users only refer to Isabelle/HOL, ignoring the other object-logics and the facilities to define new ones. Nevertheless, Isabelle/HOL benefits from the generic framework, which provides a cleaner view on general logical concepts than the more specialized implementations of the original HOL family. Isabelle generally appears slightly less cryptic to its users. Separate

concrete syntax for theory specifications has been provided early [Paulson and Nipkow, 1994]; proof scripts have become more and more stylized as well, using a few generic tactics (parameterized by theorems) instead of a large collection of special invocations. Worldwide Isabelle/HOL users have been able to conduct many significant applications over the past few years. Presently the biggest one is probably the formalization of the Java programming language by the Isabelle/Bali project [Oheimb, 2001] [Bali]. Further "official" examples and significant applications are included in [Isabelle library] (which also covers other object-logics than Isabelle/HOL).

The standard way of formal reasoning in Isabelle resembles the tactical backwards style of HOL and Coq. Some past experiments on improved presentations [Simons, 1996] [Simons, 1997] have covered a literate programming view for theories and proof scripts, and special tactics to support idioms of calculational reasoning (following the approach of [Dijkstra and Scholten, 1990]).

PVS

PVS [Owre *et al.*, 1996] (distributed by the SRI) is advertised as a tightly integrated environment for specification, proof checking, and model checking. Its most prominent features are predicate subtypes, a collection of well-integrated algebraic decision procedures, and an easily accessible user-interface for interactive theory and proof development.

The logic of PVS is usually presented as another version of "higher-order logic", although it considerably deviates from the one of HOL. In particular, HOL's distinctive view on schematic polymorphism and semantic type definitions is unavailable in PVS. In fact, the PVS logic is better understood as a version of set-theory, where certain aspects of set-membership reasoning have been singled out as a specific concept of "predicate subtypes". Type checking conditions (TCCs) are extracted and solved automatically, although the user needs to interact in difficult cases (by means of ordinary PVS proof tools). The resulting discipline approximates the casual treatment of typing in informal mathematics reasonably well. Furthermore, there is specific notation for subtypes of Cartesian products and function spaces, which are presented as "dependent types", analogous to Σ and Π in real type theories.

PVS offers powerful definitional mechanisms for algebraic datatypes and well-founded recursive functions. These have been based on set-theoretic principles according to a later paper on the "official" PVS semantics [Owre and Shankar, 1997]. The handsome integration of algebraic proof tools (including arithmetic semi-decision procedures) enables users to "grind" many everyday proof problems, without demanding much insight into logical details. PVS also provides a language of "strategies" that resembles the tactical ones of HOL, Coq, or Isabelle, but does not admit arbitrary programming or proof search.

The PVS implementation is monolithic, consisting of a large body of LISP code. The sources are not generally available, although interested parties may take

a look at the SRI (and sometimes even change a few details). The advanced proof tools and specification mechanisms are hardwired, without full reduction to basic logical concepts inside. Over the years, seasoned users of PVS have encountered a number of serious problems in practice, not just soundness issues of proving false results, but also unexpected failures. The known soundness bugs of PVS are not considered a real problem by its proponents. The focus of PVS has been changed from a "Prototype Verification System" to a tool for finding errors in formal models of software and hardware systems, which is actually falsification instead of verification. Indeed, PVS has been quite successful in this respect, attracting a considerable number of users lately.

PVS shows how far the paradigm of interactive theorem proving may get to the pragmatic side of "computer-aided verification". Most users only have a marginal interest in formal logic and proofs themselves. So far there has been rather little interest in human-oriented representations of proofs in PVS.

Mizar

Mizar [Rudnicki, 1992] [Trybulec, 1993] has emerged from a project on program verification for Algol in the 1970's (both "Algol" and "Mizar" are Arabic names for certain stars). At some point it was felt that a reasonably body of mathematical background theories are required before being able to verify actual programs. The main focus of the Mizar project has shifted towards further development of the enormous [Mizar library], while the Mizar system itself has changed very little recently. New library entries are periodically published in the "Journal of Formalized Mathematics".

The most notable aspect of Mizar is its structured proof language, which has been designed to represent common mathematical proof patterns in a formal setting. The Mizar language is tightly integrated with its particular logical background, namely classical first-order logic with an axiomatic basis of typed set-theory (according to Tarski-Grothendieck), some special support for "second order" schemas (e.g. induction), and a particular notion of mathematical structures. The proof language provides separate elements to cover proof principles from raw first-order logic, e.g. universal introduction, existential introduction and elimination (two versions), and disjunction elimination by cases. Furthermore, there is a builtin notion of "obvious" reasoning steps in order to finish terminal situations. The latter also covers first-order steps that lack a separate proof language element (e.g. universal elimination and disjunction introduction).

According to its authors, Mizar is "notorious for lack of documentation". New users are typically instructed directly by Mizar experts. Some partial documentation has eventually become available [Muzalewski, 1993]. The more detailed overview of [Wiedijk, 1999] provides an approximation of the main Mizar proof language elements in terms of plain natural deduction. The full details of Mizar proof processing have not been published so far; even the sources of the implementation are unavailable.

Apparently, Mizar represents a rather different tradition of theorem proving than the mainstream tactical systems (HOL, Coq, Isabelle, PVS etc.), with respect to the logic, the proof language, and the system architecture. The Mizar project has been very successful in building up a large body of machine-checked mathematical theories. On the other hand, Mizar also has some inherent limitations, mostly due to its "closed" approach. For example, there is no practical way to add new proof tools (say a flexible rewriting engine), or provide new specification mechanisms (say inductive sets and recursive functions). Consequently, many advanced concepts need to be simulated directly in the text by existing Mizar elements: rewriting is typically expressed by long chains of single equational reasoning steps, and inductive definitions are constructed manually on top of primitive set-theoretic concepts over and over again [Mizar library].

The structured proof language of Mizar is the main communication format between the user and the machine, and also between users themselves (e.g. when composing new theories based on existing ones). Nevertheless, the default view of the WWW presentation of [Mizar library] omits proofs. There have also been some past experiments on rendering Mizar texts in natural language [Bancerek and Carlson, 1993], but this output format is rarely encountered in practice.

1.2.2 Experiments on human-readable proofs

The relative success of flexible tactical theorem provers on the one hand, and structured mathematical proofs in Mizar on the other hand have stimulated some further research on human-readable proofs in recent years. This has eventually resulted in several experimental systems that focus on accessible representations of formal proofs themselves.

The "Mizar mode for HOL" [Harrison, 1996b] provides an alternative interface for interactive proof composition in HOL (notably HOL-Light [Harrison, 1996a]), transferring useful ideas from the Mizar proof language into the tactical setting of HOL. Harrison introduces separate concrete syntax for structured proof commands that are translated to special tactics inside, which perform basic transformations according to natural deduction schemes of raw first order logic. Harrison also spends substantial effort on automated reasoning support, for solving "trivial" situations implicitly (the concrete procedure may be exchanged by the user). The Mizar mode also covers a calculational reasoning style, which refers to a collection of mixed transitivity rules declared in the context (of $=/</\leq$ or similar relations). The system has been sufficiently developed to conduct some example proofs from classical analysis, covering a few pages of text; it has not been applied any further, though.

DECLARE [Syme, 1997a] [Syme, 1998] is a stand-alone prototype system for "declarative" proof development, which acts like a compiler for formal documents consisting of theory specifications and structured proof outlines. The proof language is based on three main principles, namely "first-order decomposition and enrichment", "second-order schema application", and "appeals to

automation". DECLARE has been advertised as "three tactic theorem proving" [Syme, 1999]. The system draws from the general experience of the HOL family (and Harrison's Mizar mode), but renounces established principles like full reduction to basic logical principles inside. DECLARE has been successfully applied by its author in some significant case-studies on Java type-safety and operational semantics [Syme, 1998]. In fact, many concepts of DECLARE have been specifically designed towards such typical applications of language modeling, with particular support for inductive definitions and proof schemes. DECLARE did not aim at more general applications, and has not been evaluated any further in practice (the system is not publicly available).

The "Structured Proof Language" (SPL) [Zammit, 1999a] [Zammit, 1999b] aims at providing another interface for proof construction in mainstream HOL, drawing from general Mizar ideas and the experience with Harrison's Mizar mode. SPL has been intended for larger scale applications, just like DECLARE, but is more careful to stay within the logical foundations of HOL. All high-level concepts of SPL are reduced to primitive HOL tactics. Zammit also spends significant effort on powerful first-order proof tools in HOL, in order to support reasoning in large steps. Another focus is on implicit simplifications (via rewriting). The SPL/HOL system has been evaluated by its author by formalizing some portions of group theory, attempting to achieve the same level of "abstraction" encountered in the informal proofs of a certain textbook.

"Mizar-Light for HOL-Light" [Wiedijk, 2001b] represents a minimal system experiment (implemented in 42 lines of ML) that achieves a readable view on first-order tactical proof schemes, mainly by exhibiting propositions explicitly in the text instead of implicitly in goal configurations.

Systems in the important class of "teaching tools for formal logic" often provide readable textual representations of proofs as well, although most seem to prefer graphical views. In any case, such systems are typically restricted to primitive inferences in pure logic, where users may occasionally specify their own set of rules, but advanced proof procedures are unavailable.

The teaching tool ProveEasy [Burstall, 1998] provides an interactive editor for primitive natural-deduction proof texts presented in a strictly backwards manner; the underlying structure is oriented towards the established λ-calculus view of type theory. Here the main idea is to make the types of sub-terms (i.e. propositions of local facts) visible in the text.

Tutch [Abel *et al.*, 2001] is a strictly text-oriented proof-checker intended for teaching constructive logic. The system deliberately excludes any kind of user interface, but acts like a batch-mode compiler of proof texts written in plain ASCII. Thus students are encouraged to focus on the task of actually writing proofs, rather than play with fancy interfaces. Proof steps in Tutch range from primitive natural deduction to more abstract arrangements of the "assertion level". Nevertheless, the system refrains arbitrary proof search, but implements an efficient algorithm for structured proof checking.

1.3 The Isar approach to formal proof documents

The primary subject of the present work is a particular approach to human-readable formal proof documents called "Isar", which abbreviates "Intelligible semi-automated reasoning". Isar covers the following levels of discourse.

1. A specific view on the problem space of formal proof (see §1.4 and §1.5). We shall introduce the categories of *primitive*, *primary*, and *presentation* formats of proofs. Thus we are able to identify the most basic components of our architecture, including the notion of *human-readable proof documents* that Isar places into the very center.

2. A concrete design of the *Isar proof language* as a viable basis for high-level proof texts following the general paradigm of *natural deduction* (see chapter 3). A number of additional concepts, mostly extra-logical ones, lift the underlying logical framework to a sufficiently abstract level that is adequate for human consumption.

   Particular care has been taken to keep the Isar language succinct. In fact, substantial parts of the language are defined as *derived elements* on top of simpler notions. The resulting framework is highly compositional, with a large combinatorial space of useful expressions ranging from simple idioms to advanced proof patterns (see also chapter 5 and chapter 6).

3. A system implementation called Isabelle/Isar [Wenzel, 2001a], which has been built on top of the generic natural deduction framework as provided by Isabelle/Pure [Paulson and Nipkow, 1994] (see also chapter 2). Being rooted at this generic level, common Isabelle object-logics may benefit directly from Isar without requiring any substantial changes (apart from some minor adaptations of existing theory libraries). New object-logics may be commenced by using Isar proof elements from the very start (e.g. see chapter 4 and chapter 8).

   Isabelle/HOL [Nipkow *et al.*, 2001] shall serve as the main workhorse for concrete examples to be presented later on. Such an advanced working environment demands a few further logic-specific provisions, notably proper integration with derived specification mechanisms (see also chapter 7). Taking the existing Isabelle/HOL setup as a starting point, we are able to provide viable support for "realistic" applications from mathematics and computer-science (e.g. see chapter 9 and chapter 10).

Isar aims at a truly versatile environment, with the following particular goals.

- Succinct language design, with few basic principles that may be combined freely. Maximum modularity of all language concepts.

  We shall only take the most fundamental language elements as primitive, and define further concepts as derived ones (while preserving the potential

for combined use with existing elements). Beyond this basic language layer we refrain from any further special abbreviations, but prefer simple idioms consisting of a few "words" in Isar. Generally speaking the Isar language is intended to support lively expression of formal reasoning, based on a relatively small vocabulary and some universal grammatical rules.

- Incremental proof processing, as suitable for interactive development.

  As a lesson learned from interactive tactical proving we observe that realistic development of "semi-automatic" proofs demands some experimentation by the writer. Step-wise evaluation of Isar proof texts may also enable beginning users to experiment with key logical concepts, e.g. the discharge behavior of assumptions in a particular context. From the perspective of readers, the incremental way of Isar proof processing induces some bias towards left-to-right interpretation, corresponding strictly to the order of language elements given in the text.

- Independence of particular object-logics, within the general framework of natural deduction.

  Our rationale is to cover all "mainstream" object-logics of Isabelle (FOL, ZF, HOL, HOLCF etc.), essentially by arranging the Isar concepts at the generic level of the Isabelle/Pure framework. This does not mean that "unusual" representations of object-logics benefit from Isar in the same way, though. For example, existing formalizations of linear and modal logics simulate sequent-calculus rules within the pure natural deduction framework, which would result in slightly impractical Isar proof texts.

- Independence of particular automated reasoning techniques.

  Automated proof search shall be never seen as a core issue of Isar proof processing, although existing procedures may be easily incorporated as "proof methods". The Isar proof language shall enforce a well-defined structure of proof texts, despite potentially ill-behaved proof tools involved in individual steps; proof methods may only operate on isolated portions of the main Isar proof configuration.

- Guarantee soundness by full reduction to basic logical principles.

  We intend to make actual formal proofs available in practice, which means that a reasonable form of internal proof presentation (in terms of basic logical principles) needs to be achieved eventually.

- Reduce accidental "formal noise" in common reasoning patterns, avoid unnecessary cluttering of proof texts.

  The danger of obscuring formal proof texts by irrelevant detail is ever present. Interestingly, tactical systems have occasionally been apt to let certain technical details intrude the course of reasoning performed by the user, which did not necessarily change the situation of unstructured proof scripts fundamentally. In Isar we need to be more careful, as reasoning

steps appear explicitly in the text. Adequate structured proof patterns typically demand a few subtle details to be got right. (A particularly illustrative example of successful formal-noise reduction is that of "induction with non-atomic statements", see §5.4.5).

- Provide a stable working environment that is usable by other people (apart from the original architect).

  Arriving at a realistic system is not just a matter of spending considerable efforts on mere implementation issues. Even more importantly, the very Isar concepts themselves need to be sufficiently simple and mature, providing a faithful model structured proofs. This certainly requires feedback from concrete applications conducted in Isabelle/Isar.

Isar follows a few general design principles, so the resulting framework is not just an arbitrary arrangement of certain ingredients, but acquires a distinctive style. Such slightly more philosophical underpinning certainly does have an impact on achieving our goals, although this is not always spelled out explicitly.

- Primacy of readability over writability.

  As we intend to produce human-readable proof texts eventually, we really need to take the (potentially large) audience of readers more seriously than writers (who are usually more versed in formal-logic and technical details of the proof system anyway). Composition of accessible presentations certainly does demand some effort in any case, not just in the context of formal reasoning. The task of being an *author* of Isar proof documents should not be taken lightly.

  Another consequence is that readers do not need any special tools to access proof texts, but may refer to traditional printed paper (or the "electronic paper" of PDF). In contrast, writers usually do require some specific tool support for interactive proof development.

- Refer to common principles of "sane" language designs.

  We generally draw from the standard repertoire of minor issues that have emerged over the last decades in high-level programming language design, e.g. block structure and static scoping of local variables.

- Liberality, or *abusus non tollit usus*.

  We generally prefer rather generic concepts that admit useful applications in many situations, despite a potentially pending danger of "inadequate" uses under certain circumstances. A notable instance of this principle is the flexible way that arbitrary proof methods (based on tactics inside) may be incorporated into Isar proofs. There are also a few "improper" language elements that enable Isabelle/Isar to absorb the old tactical style of Isabelle completely.

The open design of the Isar language enables proofs to be written in almost arbitrarily bad style. Nevertheless, it should be easier to compose adequate texts by default, although this requires some taste of the author.

- Separation of primitives versus policy.

  We explicitly distinguish two different aspects of Isar proof processing, namely logical primitives and the policy enforced by interpreting certain language elements. In particular, we refrain from treating the Isar language as another "calculus" itself, despite its inherent relation to formal logic. Thus we achieve a clear separation of concerns, enabling us to think about the Isar language in extra-logical categories.

1.4 Notions of proof according to Isar

The very notion of "proof" is hard to pin down exactly, depending on the context of discourse. We refrain from attempting a universal definition, but merely provide specific views on the problem space as relevant for Isar.

First of all, proofs shall be always required to be *fully formal* in the strong sense that any resulting theorems are guaranteed to be actually reduced to basic logical inferences (within a well-defined background theory). In practice, this means that proofs need to be processed mechanically by a (trusted) *proof checker* component. Nevertheless, users should not necessarily bother about the actual internal representations of proofs. (Just like ML programmers normally do not need to know about the machine-language that is executed eventually.) In fact, that low-level view would be quite counter-productive for our objective of human-readable proofs. Primitive derivations are apt to obscure the intentions of formal reasoning, which has historically made many people reject the idea of proof formalization altogether, if they have ever been exposed to it anyway.

In Isar we differentiate the following three levels of formal proof.

1. **Presentation format.**

   This is the final material given to recipients, i.e. the audience of (human) readers of proofs.

2. **Primary proofs.**

   The main communication format between the proof development system and the user, i.e. the (human) writer of proofs.

3. **Primitive representation.**

   The internal structure of basic inferences that serves as the very foundation of correct results.

Various theorem proving systems exhibit quite different ideas of proofs at these three levels. Even a single system may offer different options for these categories.

For example, the Coq system [Barras *et al.*, 1999] is based on dependently-typed λ-terms as the primitive format. The primary view is that of tactical proof scripts. Moreover, Coq provides two formats for presentation, either a pretty-printed output of the primary script, or a rendering of primitive λ-terms in natural language [Coscoy *et al.*, 1995].

The HOL system [Gordon, 1985a] [Gordon and Melham, 1993] [Gordon, 2000] provides a rather different view on the these levels of proof. Here the primitive layer consists of abstract theorem constructors of the inference kernel, according to "Correctness by Construction" by Milner. HOL offers several primary views on top, ranging from direct access to forward inferences to the goal-centered paradigm of tactical proving (users may also implement their own proof construction mechanisms). The standard presentation format of HOL provides a pretty-printed version of the sources, with some visual enhancement of mathematical symbols [Gordon and Melham, 1993].

In Isar we shall take the following particular view on these three levels of proof (see also the example in §1.5).

1. Presentation produces "formal proof documents", consisting of a beautified version of the primary sources. The Isabelle/Isar document preparation system automatically takes care of this, as a side-effect of formal proof processing. The final documents are meant to resemble traditional mathematical texts, with high-quality typesetting based on LaTeX. No attempt is made on any significant transformations of the primary text, e.g. we refrain from natural language generation. This makes the presentation layer of Isabelle/Isar appear as very thin.

2. The primary layer of Isar shall absorb our main efforts on reasonable concepts of human-readable proof texts. The formal proof language given here is designed to be ready for human consumption and machine-checking at the same time. Development of primary proofs is facilitated by fine-grained incremental interpretation of the source text, with meaningful output of intermediate states. Further user-interface support is provided by the generic Proof General environment (see also §1.5). Despite interactive development, the course of reasoning is expressed statically in the final text.

3. The primitive layer is treated abstractly in Isar, merely demanding a few basic principles as an interface for the upper language level (notably composition of facts and goals via higher-order resolution). The primary Isar interpretation process essentially "drives" these primitive inferences, but never lets the results intrude the text directly. As a consequence, the internal details of primitive proofs do not really matter, so Isar may both use Isabelle's traditional notion of "Correctness by Construction" or primitive proof terms of the meta-logic.

This particular division of the problem space of formal proof shall be now illustrated by a concrete example.

1.5 Example: the Knaster-Tarski Theorem

We consider a simple formulation of the Knaster-Tarski fixed-point theorem for complete lattices. The informal statement and proof outline is given below, following the textbook presentation of [Davey and Priestley, 1990, pages 93–94] with only minor notational changes.

> **The Knaster-Tarski Fixpoint Theorem.** Let L be a complete lattice and $f\colon L \to L$ an order-preserving map. Then $\bigsqcap \{x \in L \mid f(x) \leq x\}$ is a fixpoint of f.
>
> **Proof.** Let $H = \{x \in L \mid f(x) \leq x\}$ and $a = \bigsqcap H$. For all $x \in H$ we have $a \leq x$, so $f(a) \leq f(x) \leq x$. Thus $f(a)$ is a lower bound of H, whence $f(a) \leq a$. We now use this inequality to prove the reverse one (!) and thereby complete the proof that a is a fixpoint. Since f is order-preserving, $f(f(a)) \leq f(a)$. This says $f(a) \in H$, so $a \leq f(a)$.

This informal exposition shall merely serve as a guideline for our subsequent formal development in Isar. Despite being rather small, the example already shows many key elements of Isar proof composition.

As is often done in "realistic" proof formalizations, we specialize the statement to cover the concrete lattice of power sets only, which happens to be readily available in our background theory of Isabelle/HOL [Nipkow *et al.*, 2001]. The main ideas of the proof will still be presented faithfully; see [Wenzel, 2001b] for a similar proof within an abstract version of lattice theory.

1.5.1 Presentation format: typeset document output

The canonical proof (and theory) presentation format of Isabelle/Isar resembles traditional mathematical documents, either printed on paper or in a simple browsable format using PDF. Such documents are meant to be accessible to readers at large, without requiring any sophisticated tools. Some understanding of the formal languages encountered here is required, though, both the basic logic and Isar proof elements.

The subsequent Knaster-Tarski proof is based on very simple facts of set-theory only, using some lattice properties of general intersection "$\bigcap$". Note that "$\bigwedge$" stands for universal quantification; the remaining logical notation is fairly standard. The concrete syntax of Isar proof elements should at least admit the text to be read aloud, even without an exact idea about the formal semantics.

theorem *Knaster-Tarski*: $(\bigwedge x\ y.\ x \subseteq y \Longrightarrow f\ x \subseteq f\ y) \Longrightarrow \exists\, a.\ f\ a = a$
proof
  assume *mono*: $\bigwedge x\ y.\ x \subseteq y \Longrightarrow f\ x \subseteq f\ y$
  let *?H* $= \{u.\ f\ u \subseteq u\}$
  let *?a* $= \bigcap$ *?H*
  have *ge*: *f ?a* $\subseteq$ *?a*
  proof
    fix *x* **assume** *H*: $x \in$ *?H*
    then have *?a* $\subseteq x$ **..**
    also from *H* **have** *f* $\ldots \subseteq x$ **..**
    moreover note *mono*
    finally show *f ?a* $\subseteq x$ **.**
  qed
  also have *?a* $\subseteq$ *f ?a*
  proof
    from *mono* **and** *ge* **have** *f* (*f ?a*) $\subseteq$ *f ?a* **.**
    then show *f ?a* $\in$ *?H* **..**
  qed
  finally show *f ?a* $=$ *?a* **.**
qed

The Isabelle document preparation system is able to produce high-quality output from the primary text given by the user (see also §1.5.2). Informal explanations may be included as well, which may refer to arbitrary LaTeX markup. Thus adequate presentations of fully formal theory developments become readily available, leaving behind the unappealing typewriter style that still persists in many theorem provers. Formal developments do not have to look ugly!

1.5.2 Primary proof: human-readable source

The format of *primary proofs* is what the Isabelle/Isar system uses directly for input. Below we exhibit this "real source" of the same Knaster-Tarski proof.

```
theorem Knaster_Tarski:
  "(\<And>x y. x \<subseteq> y \<Longrightarrow> f x \<subseteq> f y)
    \<Longrightarrow> \<exists>a. f a = a"
proof
  assume mono:
    "\<And>x y. x \<subseteq> y \<Longrightarrow> f x \<subseteq> f y"
  let ?H = "{u. f u \<subseteq> u}"
  let ?a = "\<Inter>?H"
  have ge: "f ?a \<subseteq> ?a"
  proof
    fix x assume H: "x \<in> ?H"
    then have "?a \<subseteq> x" ..
    also from H have "f \<dots> \<subseteq> x" ..
```

```
    moreover note mono
    finally show "f ?a \<subseteq> x" .
  qed
  also have "?a \<subseteq> f ?a"
  proof
    from mono and ge have "f (f ?a) \<subseteq> f ?a" .
    then show "f ?a \<in> ?H" ..
  qed
  finally show "f ?a = ?a" .
qed
```

Apparently, the above Isar source is not far removed from the presentation format given before (§1.5.1). The raw text lacks highlighted keywords, proper printing of mathematical symbols, and contains additional quotation marks (which are required in Isabelle to delimit the inner syntax of types and terms from the primary theory and proof language), but the key structure of Isar proofs is already present.

Incidently, Isabelle/Isar sources somewhat resemble (stylized) LaTeX input. In fact, Isabelle/Isar and LaTeX share the basic idea of producing typeset documents from decent textual descriptions, with the big difference that LaTeX does not perform any formal checking, of course.

In reality, users need not directly work with raw ASCII texts as shown above, although this is possible in principle. Additional conveniences are provided by the generic Proof General environment [Aspinall, 2000] [Proof General], which essentially provides an interface for automatic *cut-and-paste* (including *undo* operations) between the source text and the underlying prover process. Proof General has been built around the XEmacs editing environment, including the X-Symbol package to take care of mathematical symbols.

Several provers are supported by Proof General, such as Coq, LEGO, PhoX, Plastic, traditional Isabelle (with the old ML top-level), and Isabelle/Isar (both for structured proof texts and proof script emulation). A typical Proof General session for Isabelle/Isar is shown in figure 1.1.

There are two main views: "script" and "proofstate", which we prefer to call *static proof text* and *dynamic proof state* in Isabelle/Isar. The former presents the source with some visual enhancements, including an indication of the proof text processed so far (which is marked as read-only in the editor in order to ensure consistency with the state of the prover process). The remaining unprocessed text may be manipulated by standard editing means of XEmacs, until the system is told to step over it by continued formal checking.

The second window provides feedback on the present Isar interpreter configuration, probably providing some clues to users on how to proceed, or figure out problems. Nevertheless, the dynamic state is significantly less important in structured proof texts than in unstructured scripts. Isar proof development really means to work on the primary text under construction, with some occasional peeks at the results achieved so far (including facts and goals).

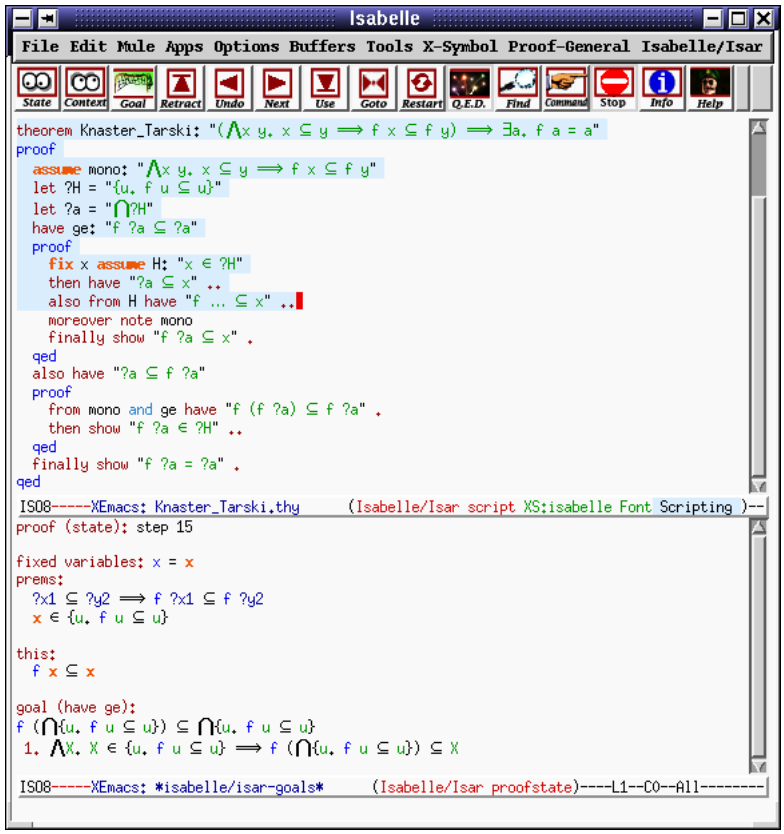

Figure 1.1: Interactive development with Proof General

Strictly speaking, such a user-interface view of the primary Isar source is already another "presentation" issue, although an even more degenerate one than the document preparation system covered before (§1.5.1). In fact, only little structure of Isar proof texts is exploited by Proof General, which has been intended as a generic front-end for existing interactive provers with unstructured scripts. For example, there is no support for actual hierarchic editing of proof texts, which Isar would easily admit due to separate checking of sub-proofs.

Independently of user-interfaces and development tools, the raw ASCII input of Isar is relevant for long-term integrity of formal proof developments. By retaining a human-readable format at the primary level, proof texts may be kept "alive" more easily, even if some of the present system components become unavailable eventually (Proof General, XEmacs, X-Symbol etc.). For example, losing X-Symbol could be amended by switching back to plain ASCII (replacing "`\<Longrightarrow>`" by "`==>`" etc.).

Such casualties do happen in reality, as may be seen from the history of Mizar [Rudnicki, 1992] [Trybulec, 1993]. Many years ago, Mizar has been tied to the now obsolete PC font (to exploit special symbols). Further development of Mizar has ever since been encumbered by the seemingly trivial issue of proper character encoding.

From a more philosophical perspective, the primary source format of Isar has the important virtue to confer meaningful formal content, even without the actual proof processor at hand. In contrast, traditional tactic scripts tend to be a one-way road only: once that existing (informal) material has been presented to the system, it has become essentially inaccessible at large, except for the original proof checker. Further derivative work in a slightly different context would typically require to go back to the informal literature, provided that can be still figured out. With inaccessible sources, there is always a pending danger of losing the results of past formalization efforts!

Certainly, the aspect of adequate archiving of theory sources becomes only relevant after formalized mathematics has been more widely accepted in practice.

1.5.3 Primitive format: internal proof terms

The Isar proof processor inherits any primitive notion of formal proofs directly from the generic Isabelle/Pure framework. Traditional "secure derivations" of the Isabelle inference kernel (due to Milner's "Correctness by Construction") are hard to visualize, though, since they only exist as an idea outside of the run-time environment of the system implementation. Instead we show the forthcoming alternative proof term format of Isabelle [Berghofer and Nipkow, 2000], which is based on typed λ-calculus (this requires Isabelle2001 or later).

$$
\begin{array}{l}
\boldsymbol{\lambda}\,(H : \bigwedge x\ y.\ x \subseteq y \Longrightarrow f\,x \subseteq f\,y).\\
\quad \mathit{HOL.exI} \cdot \lambda x.\ f\,x = x \cdot \bigcap\{u.\ f\,u \subseteq u\} \cdot\\
\quad\quad (\mathit{Ord.order.order\text{-}antisym} \cdot f\ (\bigcap\{u.\ f\,u \subseteq u\}) \cdot \bigcap\{u.\ f\,u \subseteq u\} \cdot\\
\quad\quad\quad (\mathit{subset.Inter\text{-}greatest} \cdot \{u.\ f\,u \subseteq u\} \cdot f\ (\bigcap\{u.\ f\,u \subseteq u\}) \cdot\\
\quad\quad\quad\quad \boldsymbol{\lambda}\,(X :: 'a\ \mathit{set}).\\
\quad\quad\quad\quad\quad \boldsymbol{\lambda}\,(\mathit{Ha} : X \in \{u.\ f\,u \subseteq u\}).\\
\quad\quad\quad\quad\quad\quad \mathit{Calculation.order\text{-}subst2} \cdot \bigcap\{u.\ f\,u \subseteq u\} \cdot X \cdot \lambda x.\ f\,x \cdot X \cdot\\
\quad\quad\quad\quad\quad\quad\quad (\mathit{subset.Inter\text{-}lower} \cdot X \cdot \{u.\ f\,u \subseteq u\} \cdot \mathit{Ha})\\
\quad\quad\quad\quad\quad\quad\quad \cdot (\mathit{HOL.iffD1} \cdot X \in \{x.\ f\,x \subseteq x\} \cdot f\,X \subseteq X \cdot\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{Set.mem\text{-}Collect\text{-}eq} \cdot X \cdot \lambda u.\ f\,u \subseteq u)\\
\quad\quad\quad\quad\quad\quad\quad\quad \cdot \mathit{Ha})\\
\quad\quad\quad\quad\quad\quad\quad \cdot H)\\
\quad\quad\quad\quad \cdot (\mathit{subset.Inter\text{-}lower} \cdot f\ (\bigcap\{u.\ f\,u \subseteq u\}) \cdot \{u.\ f\,u \subseteq u\} \cdot\\
\quad\quad\quad\quad\quad (\mathit{HOL.iffD2} \cdot f\ (\bigcap\{u.\ f\,u \subseteq u\}) \in \{x.\ f\,x \subseteq x\} \cdot\\
\quad\quad\quad\quad\quad\quad f\ (f\ (\bigcap\{u.\ f\,u \subseteq u\})) \subseteq f\ (\bigcap\{u.\ f\,u \subseteq u\})\\
\quad\quad\quad\quad\quad\quad \cdot (\mathit{Set.mem\text{-}Collect\text{-}eq} \cdot f\ (\bigcap\{u.\ f\,u \subseteq u\}) \cdot \lambda u.\ f\,u \subseteq u)\\
\quad\quad\quad\quad\quad\quad \cdot (H \cdot f\ (\bigcap\{u.\ f\,u \subseteq u\}) \cdot \bigcap\{u.\ f\,u \subseteq u\} \cdot\\
\quad\quad\quad\quad\quad\quad\quad (\mathit{subset.Inter\text{-}greatest} \cdot \{u.\ f\,u \subseteq u\} \cdot f\ (\bigcap\{u.\ f\,u \subseteq u\}) \cdot\\
\quad\quad\quad\quad\quad\quad\quad\quad \boldsymbol{\lambda}\,(X :: 'a\ \mathit{set}).
\end{array}
$$

$$
\begin{array}{l}
\boldsymbol{\lambda}\,(Ha : X \in \{u.\ f\ u \subseteq u\}).\\
\quad \mathit{Calculation.order\text{-}subst2} \cdot \bigcap\{u.\ f\ u \subseteq u\} \cdot X \cdot \lambda x.\ f\ x\\
\qquad \cdot X\\
\qquad \cdot\ (\mathit{subset.Inter\text{-}lower} \cdot X \cdot \{u.\ f\ u \subseteq u\} \cdot Ha)\\
\qquad \cdot\ (\mathit{HOL.iffD1} \cdot X \in \{x.\ f\ x \subseteq x\} \cdot f\ X \subseteq X \cdot\\
\qquad\quad (\mathit{Set.mem\text{-}Collect\text{-}eq} \cdot X \cdot \lambda u.\ f\ u \subseteq u)\\
\qquad\quad \cdot\ Ha)\\
\qquad \cdot\ H)))))
\end{array}
$$

We see that most of the primary proof structure has been lost after reduction to primitive concepts. For example, the local result of "**have** *ge*: $f\ ?a \subseteq ?a$" (internally $f\ (\bigcap\{u.\ f\ u \subseteq u\}) \subseteq \bigcap\{u.\ f\ u \subseteq u\}$) is used twice in the Isar text, and appears in two independent copies in the primitive proof due to internal β-normalization. Another problem is posed by the seemingly trivial issue of adequate naming of bound variables, due to arbitrary α-conversion inside.

It would indeed be hard to recover a readable Isar text from the primitive representation, even though Knaster-Tarski is still a very simple example. Note that we intend to cover much larger applications as well. In fact, this is the deeper reason why Isar takes high-level texts as a starting point, and produces low-level proof representations via interpretation from top to bottom.

1.6 Overview of the thesis

1.6.1 Part I: Foundations

The main objective of Isar foundations is to turn existing formal-logic concepts into a viable language environment for natural deduction proof texts, without requiring extensive theoretical studies first. Isar particularly draws from known principles of natural deduction reasoning in minimal higher-order logic, with specific support for higher-order resolution and higher-order unification (chapter 2). The Isar proof language itself provides a qualitatively different view, following general concepts of high-level programming languages and leaving behind raw logic. These two levels of discourse are bridged by the Isar/VM interpreter (chapter 3). The basic structure of natural deduction proof texts is explored by the example of pure first-order logic (chapter 4).

1.6.2 Part II: Techniques

The generic Isar framework has substantial potential for "advanced" techniques of formal proof composition, beyond raw natural deduction. We give a systematic exposition of practically relevant Isar proof patterns, including derived elements like generalized elimination, cases and induction (chapter 5). The important paradigm of calculational reasoning (within natural deduction) is explored as well (chapter 6). All of these techniques have been distilled from concrete Isabelle/Isar applications, and have already proven viable in practice.

1.6.3 Part III: Applications

Isabelle/Isar is able to cover a broad range of applications. We include concrete examples from pure logic (chapter 8), mathematics (chapter 9), and computer-science (chapter 10). The latter two make use of the Isabelle/HOL application environment (chapter 7), which gives rise to some further logic-specific Isar proof techniques. As a general rule, we never "explain" concrete proofs informally, but let the formal Isar text stand on its own. Nevertheless, specific Isar proof techniques may well be discussed separately. All formal theory developments are given complete and unabridged, so the included applications provide evidence for "realistic" Isabelle/Isar proof documents (as produced with the official version of Isabelle99-2 from February 2001).

Part I

Foundations

Chapter 2

Preliminaries

We briefly review a few foundational issues that are relevant to the Isar framework to be introduced later on. This includes basic mathematical notions, and an abstract model for generic natural deduction based on minimal higher-order logic. The latter eventually leads to a viable environment for primitive logical inferences due to the Isabelle tradition, with the notable inclusion of fundamental tool support via higher-order unification and back-chaining. Some details of the existing view have been simplified and generalized for the purposes of Isar.

2.1 Basic mathematical notions

We outline the main aspects of our *semi-formal* background language for traditional "pen-and-paper" treatment of mathematical concepts. We basically employ a standard version of classical set theory, using common mathematical notation as far as possible, with some bias towards conventions of higher-order functional programming (according to Haskell or ML) and higher-order logic as in Isabelle/HOL [Nipkow *et al.*, 2001] (see also chapter 7). Although many of the *formal* logical elements to be introduced later on (such as λ-calculus and higher-order logic) will share substantial parts of the notation introduced here, these are still different levels of discourse with quite different formal status.

Sets. Some basic sets are taken for granted: truth values $bool = \{\mathsf{true}, \mathsf{false}\}$ and natural numbers $nat = \{0, 1, 2, \ldots\}$. Common set constructions like comprehension $\{x \in A.\ P\ x\}$, power sets *set of* A and *finite set of* A, Cartesian products $A \times B$ and disjoint sums $A \mid B$ are available as well. We closely stick to standard set theory notation for further operations, such as $x \in A$, $A \cup B$, $A \cap B$, $A - B$.

Compound expressions. Common functional programming notation is used for conditional expressions $\mathsf{if}\ b\ \mathsf{then}\ x\ \mathsf{else}\ y$ (where b may be a proposition

or boolean value), as well as let $x_1 = y_1$; ...; $x_n = y_n$ in $e[x_1, \ldots, x_n]$ which abbreviates $e[y_1, \ldots, y_n]$.

Vector notation. Vectors are finite sequences of elements treated as a separate notational device. We write $\vec{a}$ for the vector of elements $a_1, \ldots, a_n$. A single element x may be identified with a singleton vector $\vec{x}$. Vectors may be appended by juxtaposition: $\vec{x}\ \vec{y} = x_1, \ldots, x_m, y_1, \ldots, y_n$.

Lists. A^* shall denote the set of finite lists over a given set A. Lists are built up inductively from the empty list [] ("nil") and $x \circ xs$ ("cons") for $x \in A$ and $xs \in A^*$. We write $[x_1, \ldots, x_n]$ to denote the list $x_1 \circ \cdots \circ x_n \circ []$ (the cons operator is nested to the right). The append operation is defined as $[x_1, \ldots, x_m] \mathbin{@} [y_1, \ldots, y_n] = [x_1, \ldots, x_m, y_1, \ldots, y_n]$. The *flat* function iteratively appends lists of lists: $\mathit{flat}\ [xs_1, \ldots, xs_n] = xs_1 \mathbin{@} \ldots \mathbin{@} xs_n$. The difference of lists $xs - ys$ means to remove individual occurrences of members of ys from xs (from left to right), in particular $(as \mathbin{@} bs) - as = bs$.

The *map* combinator lifts a function to operate on lists, i.e. $\mathit{map}\ f\ [x_1, \ldots, x_n] = [f\,x_1, \ldots, f\,x_n]$. The *iterate* operator generalizes *map* by maintaining an additional result: $\mathit{iterate}\ f\ (x_0, [a_1, \ldots, a_n]) =$ let $(x_1, b_1) = f\ (x_0, a_1)$; ...; $(x_n, b_n) = f\ (x_{n-1}, a_n)$ in $(x_n, [b_1, \ldots, b_n])$.

$A^+ = A^* - \{[]\}$ shall denote the set of non-empty lists over A. Functions *first* and *last* defined on A^+ shall select the first and last elements, respectively.

Functions. Let $A \to B$ denote the set of total functions from sets A to B, and $A \rightharpoonup B$ denote the set of partial functions (which share the same notation as total ones). Following common practice "$f \in A \to B$" is written "$f\colon A \to B$". As usual in set theory, functions are identified with their graph; thus we may also use plain set notation, e.g. $\{\}$ for the completely undefined function.

We use λ-notation $\lambda x \in A.\ f(x)$ to refer to the function mapping any $x \in A$ to $f(x)$. Function application is simply written as $f\,x$, omitting parentheses as far as possible. Both abstraction and application may be iterated: $\lambda\vec{x}.\ f(\vec{x}) = \lambda x_1.\ \ldots\ \lambda x_n.\ f(\vec{x})$ and $f\ \vec{x} = (\ldots\ (f\ x_1)\ \ldots)\ x_n$. An alternative notation for application is $x \triangleright f$, which may be pronounced as "feed x into f"; the $\triangleright$ operator is left-associative and binds strongly (but weaker than plain application).

Point-wise update of functions is written in postfix notation, using $f(x := y)$ to denote the function mapping x to y and any other a to $f\,a$. The special notation $f(x := \mathit{undefined})$ means to delete an entry, i.e. $f - \{(x, f\,x)\}$. Iterated update $f\ {+\!\!+}\ g$ of with a collection g of pairs (x, y) is defined in the obvious manner, for g being either a list (counted from left to right) or a partial function.

Left-to-right sequential composition of functions f and g is written as $f;\ g$, which is defined as $(f;\ g)\ x = g\ (f\,x)$. The dual notation $g \circ f$ for right-to-left composition is available as well.

Procedures. Let A and B be sets. A partial function $s\colon \mathit{nat} \rightharpoonup B$ is called a *sequence* iff $\forall\, i.\ s\ i\ \mathit{undefined} \longrightarrow (\forall\, j.\ i < j \longrightarrow s\ j\ \mathit{undefined})$, i.e. once that

an undefined position is encountered only undefined positions may follow. A sequence is infinite iff it is a total function $nat \to B$.

Let *canonical* $s = s\ 0$ refer to the head element of a sequence, which is considered the "canonical" one in the denumeration. Furthermore, let *truncate* $s\ i = s\ i$ for $i = 0$ and undefined for $i > 0$, i.e. *truncate* restricts a sequence to its canonical result. Finite sequences coincide with lists; we extend the append operation on sequences accordingly, such that $s_1 \mathbin{@} s_2 = s_1$ for s_1 being infinite. The *flat* operations on lists of lists is transferred to sequences of sequences analogously.

A function $p\colon A \to (nat \rightharpoonup B)$ is called a *procedure* iff any $p\ x$ is a sequence (for $x \in A$). We write $A \to^{**} B$ for the set of procedures from A to B. Note that procedures need not necessarily be computable functions. Procedures $p\colon A \to (nat \rightharpoonup B)$ and $q\colon B \to (nat \rightharpoonup C)$ may be composed in a canonical fashion as follows: any result sequence $p\ x$ is mapped through q (by function composition) and the emerging sequence of sequences is flattened; consequently we define $(p;\ q)\colon A \to (nat \rightharpoonup C)$ as $(p;\ q)\ x = \mathit{flat}\ (q \circ p\ x)$. Alternative choice of procedures $p \mid q$ means to append the individual result sequences (with left-to-right preference): $(p \mid q)\ x = p\ x \mathbin{@} q\ x$.

A function $f\colon A \to B$ may be turned into a procedure $A \to^{**} B$ by replacing any $y = f\ x$ (for $x \in A$) by a singleton sequence with canonical result y. Furthermore, a procedure may be converted back into a (partial) function by truncating each individual result sequence. In order to avoid excessive detail later on, we usually treat procedures and (partial) functions uniformly, assuming that implicit conversions are inserted as required. In particular, this convention admits to refer to complex operations succinctly in functional expressions (e.g. higher-order unification which enumerates all possible solutions).

Records. Tuple structures with explicitly labeled fields are expressed in a concise manner by using record notation. In reminiscence of ordinary tuples $(x_1, \ldots, x_n) \in A_1 \times \cdots \times A_n$, let $(\!|a_1 \in A_1, \ldots, a_n \in A_n|\!)$ denote the set of records over fields $\vec{a}$ with values from $\vec{A}$, and write individual record expressions as $(\!|a_1 = x_1, \ldots, a_n = x_n|\!)$. To accommodate large record specifications we also use the declaration format **record** $R = a_1 :: A_1 \ \ldots\ a_n :: A_n$.

For any record R with some field $a \in A$ the following standard operations are available: field selection *get-a*: $R \to A$, field update *put-a*: $A \to R \to R$, and the functional *map-a*: $(A \to A) \to (R \to R)$ for lifting field operations to records, which is defined as $\mathit{map\text{-}a}\ f = \lambda r.\ \mathit{put\text{-}a}\ (f\ (\mathit{get\text{-}a}\ r))$.

2.2 Minimal Higher-Order Logic

We briefly outline simply-typed minimal higher-order logic, which shall serve as the very basis for formal-logic issues to be covered later on. The subsequent presentation draws from similar formulations of the generic framework underlying Isabelle/Pure [Paulson, 1989] [Paulson, 1990], with further influences of type theory presentations like [Barendregt and Geuvers, 2001].

2.2.1 Types and terms

The basic syntactic framework of the logical environment introduced below is that of simply-typed λ-terms modulo $\alpha\beta\eta$-conversion, following the established practice of higher-order abstract syntax [Pfenning and Elliott, 1988].

Let *name* be a globally fixed (infinite) set of names, e.g. the set of strings over a finite alphabet. Subsequently, we use implicit "copies" of *name* to achieve separate naming of various syntactic categories (variables, constants, etc.).

Types τ are inductively defined as first-order "term" structures: $\tau = ?\alpha \mid \alpha \mid (\tau_1, \ldots, \tau_n)c$, with schematic type variables $?\alpha \in name$, fixed type variables $\alpha \in name$, and type constructors $c \in name \times nat$. Type variables may be also represented by identifiers prefixed by a prime, e.g. writing $'a$ for α. The second component of a type constructor is called its *arity*; it is usually suppressed as it is clear from the context, e.g. we write *bool* instead of (*bool*, 0). The special type constructor $(\Rightarrow, 2)$ is written as infix and nested to the right, as in the "curried" type expression $\tau_1 \Rightarrow \tau_2 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \sigma$; the latter may be also abbreviated as $\vec{\tau} \Rightarrow \sigma$. For 0-ary type constructors we write $()c$ merely as c. Let *type* be the set of all well-formed types.

Terms t are simply-typed λ-terms which are built over schematic variables $?x_\tau \in name \times type$, fixed variables $x_\tau \in name \times type$, and constants $c_\tau \in name \times type$ as follows: $t = ?x_\tau \mid x_\tau \mid c_\tau \mid \lambda x_\tau.\ t \mid t_1\ t_2$. The typing relation $t\colon \tau$ is defined inductively, with $a_\tau\colon \tau$ for atomic terms (variables and constants) and the subsequent rules for abstraction and application:

$$\frac{t\colon \sigma}{(\lambda x_\tau.\ t)\colon \tau \Rightarrow \sigma} \qquad \frac{t_1\colon \tau \Rightarrow \sigma \quad t_2\colon \tau}{(t_1\ t_2)\colon \sigma}$$

Note that this form of type assignment does not require a separate context of variable typings, since all atomic terms are already equipped with type annotations beforehand. A term t is called *well-typed* iff $\exists\, \tau.\ t\colon \tau$. Apparently, each well-typed term has a unique type. Let *term* be the set of all well-typed terms.

The usual notions of *substitution* and *instances* are taken for granted. Using postfix notation we write (simultaneous) substitution as $\tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ for types, and $t[t_1/x_1, \ldots, t_n/x_n]$ for terms (which has to respect types). Furthermore, λ-terms shall be considered equal modulo the usual equational theory of $\alpha\beta\eta$-conversion.

A *signature* Σ is a collection of declarations of type constructors (with arities) and constants (with types), such that constant declarations are closed wrt. type instances and only refer to already declared type constructors. A signature may be specified by giving schemes of type constructor arities $(\alpha_1, \ldots, \alpha_n)c$ and constant declarations $c :: \tau$.

A type is called *well-formed wrt. a signature* Σ iff it is well-formed and only refers to type constructors of Σ. Likewise, a term is called *well-typed wrt. a signature*

Σ iff it is well-typed and only refers to type constructors and constants of Σ. In practice, we refer implicitly to the standard signature of the present context.

Finally note that "fixed" versus "schematic" variables as introduced above are just separate syntactic expressions of the very same formal concept of variables. The difference is merely one of a *policy* in certain logical operations to be introduced later on (notably higher-order resolution, see §2.4) [Paulson, 1989]: schematic variables may get instantiated on the fly, while fixed ones need to be left unchanged in the present scope.

2.2.2 Propositions and theorems

Well-typed terms of the special type *prop* are called *propositions*; the set of propositions is called *prop* as well. Statements of minimal higher-order logic involve separate logical connectives of $\bigwedge$ (universal quantification) and $\Longrightarrow$ (implication). From now on, we assume that the signature of the current context contains at least the following declarations:

prop	type of propositions
$\bigwedge :: (\alpha \Rightarrow \mathit{prop}) \Rightarrow \mathit{prop}$	universal quantifier (binder)
$\Longrightarrow :: \mathit{prop} \Rightarrow \mathit{prop} \Rightarrow \mathit{prop}$	implication (right-associative infix)

The common binder notation $\bigwedge x_1 \cdots x_n.\ \varphi$ refers to nested application of universal quantifiers and abstractions $\bigwedge(\lambda x_1.\ \cdots \bigwedge(\lambda x_n.\ \varphi))$. "Curried" implication $A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow C$ is occasionally abbreviated as $\vec{A} \Longrightarrow C$.

The set *theorem* is defined inductively as a certain subset of derivable "sequents" from (*finite set of prop*) $\times$ *prop*. We write $\Gamma \vdash \varphi$ for $(\Gamma, \varphi) \in \mathit{theorem}$, and write $\vdash \varphi$ for $\{\} \vdash \varphi$. The subsequent inductive definition of $\Gamma \vdash \varphi$ depends on a fixed set of propositions (also called *axioms*) which is required to be closed wrt. type instantiation.

$$\frac{(\text{if } \varphi \text{ is an axiom})}{\vdash \varphi}\ (\mathit{axiom}) \qquad \frac{}{\{\varphi\} \vdash \varphi}\ (\mathit{assumption})$$

$$\frac{\Gamma \vdash \psi}{\Gamma - \{\varphi\} \vdash \varphi \Longrightarrow \psi}\ (\Longrightarrow\text{-}\mathit{intro}) \qquad \frac{\Gamma_1 \vdash \varphi \Longrightarrow \psi \quad \Gamma_2 \vdash \varphi}{\Gamma_1 \cup \Gamma_2 \vdash \psi}\ (\Longrightarrow\text{-}\mathit{elim})$$

$$\frac{\Gamma \vdash \varphi \quad (\text{if } x \text{ not free in } \Gamma)}{\Gamma \vdash \bigwedge x.\ \varphi}\ (\bigwedge\text{-}\mathit{intro}) \qquad \frac{\Gamma \vdash \bigwedge x.\ \varphi}{\Gamma \vdash \varphi[t/x]}\ (\bigwedge\text{-}\mathit{elim})$$

An alternative presentation of these rules is given below, according to common inference notation for natural deduction (e.g. see the exposition in [Thompson, 1991] or [Basin and Matthews, 2001]). Local contexts involved in the rules are

treated implicit here; the *axiom* and *assumption* schemes are suppressed.

$$\frac{\begin{array}{c}[\varphi]\\ \vdots\\ \psi\end{array}}{\varphi \Longrightarrow \psi}\ (\Longrightarrow\text{-}intro) \qquad \frac{\varphi \Longrightarrow \psi \quad \varphi}{\psi}\ (\Longrightarrow\text{-}elim)$$

$$\frac{\begin{array}{c}[x]\\ \vdots\\ \varphi\end{array}}{\bigwedge x.\ \varphi}\ (\bigwedge\text{-}intro) \qquad \frac{\bigwedge x.\ \varphi}{\varphi[t/x]}\ (\bigwedge\text{-}elim)$$

To achieve succinct presentations later on, logical inference rules are occasionally treated like functions (taking scheme parameters or premises as arguments), e.g. *assumption* $\varphi = \{\varphi\} \vdash \varphi$. Also note that meta-level theorems (especially those of non-atomic statements involving $\bigwedge/\Longrightarrow$) are occasionally called "rules" as well. This liberal terminology makes some sense, because theorems give rise to canonical inference rules via higher-order back-chaining (see also §2.4).

A *theory* Θ consists of a signature Σ (cf. §2.2.1) plus a set of axioms (closed wrt. type instantiation). A theory may be specified by giving declarations for the signature part and stating axioms $\vdash \varphi$. We usually refer implicitly to the standard theory (and signature) of the present context.

As a general convention (following [Paulson and Nipkow, 1994]), free variables occurring in theorems presented at the top-level theory context shall be considered as implicitly generalized. This may be expressed by replacing fixed variables x (for terms) and α (for types) by schematic ones $?x$ and $?\alpha$. Outermost quantification "$\bigwedge x$" achieves an equivalent effect, but does not work for type variables. So $\vdash A \Longrightarrow A$ may be read as $\vdash ?A \Longrightarrow ?A$ or $\vdash \bigwedge A.\ A \Longrightarrow A$.

The inference system given above supports *schematic polymorphism*, in the sense that arbitrary type instances of theorems are guaranteed to be derivable as well (which requires well-typedness of terms and propositions to be preserved in the first-place). The deeper reason for this is that both the declarations of constant schemes and axioms are closed by type instantiation. The following (admissible) rule captures schematic polymorphism succinctly; it is quite easy to establish by induction over derivations.

$$\frac{\Gamma \vdash \varphi \quad (\text{if } \alpha \text{ not in } \Gamma)}{\Gamma \vdash \varphi[\tau/\alpha]} \qquad \text{or:} \qquad \frac{\begin{array}{c}[\alpha]\\ \vdots\\ \varphi\end{array}}{\varphi[\tau/\alpha]}$$

Minimal higher-order logic considered so far is sufficiently expressive to represent further standard logical connectives ($\exists$, $\wedge$, $\vee$, $\neg$ etc.) directly within the existing

system. For example, $\exists\, x.\ P\ x$ may be represented according to its canonical elimination form as $\bigwedge C.\ (\bigwedge x.\ P\ x \Longrightarrow C) \Longrightarrow C$ (see also chapter 8).

Immediate extensions like this are not the primary intention of the pure logical framework, though. An actual working environment like Isabelle/HOL (see chapter 7) is embedded as an *object-logic* instead. This involves separate axiomatization of a "derivability judgment" that coerces object-level statements to meta-level propositions. Isabelle [Paulson and Nipkow, 1994] traditionally uses the functional constant *Trueprop*, which is suppressed in the concrete syntax. So "$\vdash \varphi$" may actually refer to $\vdash$ *Trueprop* φ, if φ is an object-level formula.

Nevertheless, it is good to know that the pure framework is able to represent standard logical connectives directly. The Isar framework introduced later on (see chapter 3) essentially provides a reflection of minimal-logic concepts to the level of structured proof texts. The previous observation on connectives may serve as a guideline for advanced reasoning patterns (e.g. see chapter 5), like the "existential" proof context element that is based on the general idea underlying $\bigwedge C.\ (\bigwedge x.\ P\ x \Longrightarrow C) \Longrightarrow C$ (see §5.3).

Theorems routinely occur in finite collections, so we define *fact* = *theorem** as the set of lists of theorems, which shall be used wherever results of derivations arise in the present context. Technically, this serves as a (partial) replacement for multiple result sequents, as available in the slightly more complex setting of DECLARE [Syme, 1997a] [Syme, 1998] [Syme, 1999]. The immediate view of conjunction as $\bigwedge C.\ (A \Longrightarrow B \Longrightarrow C) \Longrightarrow C$ is occasionally helpful as well.

2.3 Definitional theory extensions

Theories may be extended by abstract syntax declarations (§2.2.1) and axioms (§2.2.2). Given a theory Θ, we may specify an extension Θ' as follows: $\Theta' = \Theta \cup (\vec{\alpha})c \cup c :: \tau \cup \vdash \varphi$, which is meant to introduce new type constructors, term constants, or axioms. Note that the actual end-user environment will provide a higher-level view on theory specifications, with concrete syntax for primitives (see chapter 3) as well as derived extension mechanisms (see chapter 7). In reality, only those theory extension schemes are considered "appropriate" that qualify as *definitional* ones for meta-theoretical reasons.

As a prerequisite for definitional equations expressed within the framework itself, we introduce a notion of (extensional) equality by axiomatic means. From now on, all theories shall contain the following constant and axiom declarations.

$\equiv :: \alpha \Rightarrow \alpha \Rightarrow \mathit{prop}$	equality relation (infix)
$\vdash x \equiv x$	reflexivity law
$\vdash x \equiv y \Longrightarrow P\ x \Longrightarrow P\ y$	substitution law
$\vdash (\bigwedge x.\ f\ x \equiv g\ x) \Longrightarrow f \equiv g$	extensionality
$\vdash (A \Longrightarrow B) \Longrightarrow (B \Longrightarrow A) \Longrightarrow A \equiv B$	coincidence with equivalence

Various notions of definitional extensions may now be identified as restricted axiomatizations over "$\equiv$".

2.3.1 Simple definitions

The most basic discipline of constant definitions essentially just introduces abbreviations for concrete expressions within the logic; see also [Pitts, 1993] for the HOL point of view. Let $c :: \tau$ refer to a "new" constant declaration wrt. the current theory context Θ. Furthermore, let $c_\tau \equiv t$ be a well-typed equation such that t neither contains the constant c nor any term variables, and the type variables of t are already covered by its type τ (i.e. t must not contain any "hidden" type dependencies). Then the extension $\Theta' = \Theta \cup c :: \tau \cup \vdash c_\tau \equiv t$ qualifies as a *simple definition*.

This strict form of definition enjoys a number of common meta-theoretical properties, e.g. preservation of completeness, decidability, consistency, and standard models (according to [Pitts, 1993]). The key property of simple definitions is that $\vdash \varphi$ in Θ' iff $\vdash \varphi[t/c]$ in Θ (where $\varphi[t/c]$ has all type instances of c_τ expanded by the corresponding type instance of t). A basic consequence is the important property of *syntactic conservativity*, in the sense that any theorem of the new context that is formulated in the old syntax already holds in the old context. The old syntax does not mention the constant c so conservativity follows trivially from the previous expansion property.

Simple definitions may be presented slightly more liberally without changing their meta-theory. In particular, the important special case of function definitions may be written succinctly as $\vdash c\ \vec{x} \equiv t$ instead of the raw $\vdash c \equiv \lambda\vec{x}.\ t$ (recall that "$\equiv$" is extensional).

2.3.2 Weakened definitions

Let $\Theta' = \Theta \cup c :: \tau \cup \vdash c_\tau \equiv t$ be a simple definitional extension. Any other extension $\Theta'' = \Theta \cup c :: \tau \cup \vdash \vec{\varphi}$, for $\vdash \vec{\varphi}$ being derivable in Θ', is called *weakened definition*. The most common instance of this scheme are *conditional definitions* of the form $\Theta \cup c :: \tau \cup \vdash \vec{\chi} \Longrightarrow c_\tau \equiv t$ for arbitrary conditions $\vec{\chi}$. Another useful instance are *loose specifications* $\Theta \cup c :: \tau \cup \vdash P\ c_\tau$, provided that $\vdash P\ t$ is derivable in the original context.

It is easy to see that weakened definitions still enjoy the basic properties of syntactic conservativity, and preservation of consistency and standard models as before (all consequences of Θ'' are already covered by Θ'). On the other hand, the exact correspondence of $\vdash \varphi$ versus the expansion $\vdash \varphi[t/c]$ has been lost, only the left-to-right implication holds in general. Weakened definitions turn out as a fairly liberal specification mechanism that merely happens to be "topped" by exact definitions outside of the formal context.

The two most extreme cases of weakened definitions are unspecified *constant declarations* of the form $\Theta \cup c :: \tau$ (like *arbitrary* $:: \alpha$ in Isabelle/HOL, see chapter 7 and chapter 8), and initial axiomatizations of full object-logics (all object-rules are derivable from a hypothetical definition $\vdash \mathit{Trueprop}\ A \equiv \top$).

2.3.3 Overloaded definitions

The scheme of overloaded constant definitions [Wenzel, 1997] renounces the requirement to have declarations $c :: \tau$ and definitions $c_\tau \equiv t$ agree on the type scheme τ, instead several (non-overlapping) type instances of c may be specified, e.g. $\Theta \cup c :: \alpha \cup \vdash c_{nat} \equiv 0 \cup \vdash c_{bool} \equiv \mathit{False} \cup \vdash c_{\alpha\times\beta} \equiv (c_\alpha, c_\beta)$. See [Wenzel, 1997] for further details on well-formedness conditions of overloaded definitions; the end-user view is covered in §7.1.2.

This rather liberal definition scheme offers interesting ways to specify generic operations, depending on the structure of (simple) types. It even covers "object-oriented" concepts like "method overriding" and "late-binding" [Naraschewski and Wenzel, 1998]. A slightly more conventional view on overloading is exploited by the concept of "axiomatic type classes", which offers a light-weight mechanism for abstract theories (see also §7.2.4).

Overloading does not lose any further meta-theoretical properties beyond those given up by weakened definitions already. Note that the tradition of relatively weak meta-theoretical properties goes back to the Gordon/HOL system [Gordon and Melham, 1993] [Pitts, 1993], which covers loose specifications (but no overloading). Isabelle/HOL [Nipkow *et al.*, 2001] (see also chapter 7) routinely uses overloading in its main library. Designers of different object-logics may choose to ignore such exotic features, but restrict themselves to simple definitions.

2.4 Higher-order resolution

The main purpose of minimal higher-order logic (§2.2) as a logical framework [Paulson, 1989] [Paulson, 1990] is to represent nested natural deduction rules as formulas over $\bigwedge/\Longrightarrow$. According to Paulson, the idea of extending original (first-order) natural deduction [Gentzen, 1935] to arbitrary nesting goes back to [Schroeder-Heister, 1984]. In the Isabelle framework, the presentation becomes slightly more elegant, though, since low-level syntactic notions like Skolem constants and textual inferences are recast via handsome $\bigwedge/\Longrightarrow$ connectives.

As a consequence of this particular view on minimal logic, the primitive introductions and eliminations (§2.2.2) lose some significance in practice, but get replaced by the derived concepts of higher-order resolution (for composing rules in a natural manner) and proof by assumption (for finishing a situation).

We will write $r \cdot \vec{a}$ for the resulting theorem of resolving fact $\vec{a}$ in parallel into a rule r. Resolution may indeed be read like a generalized application of λ-

calculus, but it covers implicit lifting over local contexts of $\bigwedge/\Longrightarrow$, as well as higher-order unification (see §2.4.2).

2.4.1 Hereditary Harrop Formulas

The language of $\bigwedge/\Longrightarrow$ formulas admits to represent different classes of "rules", depending on the intended kind of inference framework, see also the literature on λ-Prolog for further details [Miller, 1991].

In particular, the set of *Horn Clauses* merely consists of curried implications of atomic formulas, with a flat prefix of outer parameters. This set may be specified succinctly as $\bigwedge x^*.\ A^* \Longrightarrow A$, where x shall represent variables and A atomic propositions (not containing $\bigwedge$ or $\Longrightarrow$). Since outermost parameters are usually expressed by free variables (both in Prolog and Isabelle tradition), the presentation may be simplified to $A^* \Longrightarrow A$. This nicely corresponds to the common two-dimensional format of inference rules:

$$\frac{A_1 \quad \ldots \quad A_n}{A}$$

Propositions in *Hereditary Harrop Format* (*HHF*) [Miller, 1991] generalize such rules by admitting arbitrary nested statements as assumptions (conclusions are still atomic); we define the set H of HHF formulas inductively as follows: $H = \bigwedge x^*.\ H^* \Longrightarrow A$. Outermost parameters are usually suppressed as before; *generalize* presents the generality of a rule in terms of schematic variables.

$$\frac{\bigwedge \vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow A\ \vec{x}}{\vec{H}\ ?\vec{x} \Longrightarrow A\ ?\vec{x}}\ (\mathit{generalize})$$

HHF admits general proof schemes to be represented succinctly. For example, mathematical induction may be stated directly at the meta-level as $\vdash P\ 0 \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (\mathit{Suc}\ n)) \Longrightarrow P\ n$, instead of a typical object-level encoding like $\vdash P\ 0 \Longrightarrow (\forall n.\ P\ n \longrightarrow P\ (\mathit{Suc}\ n)) \Longrightarrow P\ n$, which is slightly awkward since $\forall/\longrightarrow$ need to be treated by explicit rule applications later. Any proposition of minimal higher-order logic may be presented in HHF normal form, because the law $\vdash (P \Longrightarrow (\bigwedge x.\ Q\ x)) \equiv (\bigwedge x.\ P \Longrightarrow Q\ x)$ allows $\bigwedge$ and $\Longrightarrow$ to be commuted such that parameters occur as a flat prefix at each level of rule nesting.

In Isabelle [Paulson and Nipkow, 1994] a *goal* is represented as a theorem, which is $\vdash \varphi \Longrightarrow \varphi$ in the beginning and gets transformed to become $\vdash \varphi$ eventually. A *tactic* is any procedure *theorem* $\rightarrow^{**}$ *theorem* that does not affect the main conclusion φ, nor the implicit assumption context. Intermediate goal configurations are of the form $\vdash \vec{\chi} \Longrightarrow \varphi$, where the *subgoals* $\vec{\chi}$ that are again HHF formula $\bigwedge \vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow A\ \vec{x}$. Here the parameters $\vec{x}$ need to be treated as "arbitrary, but fixed", while the premises $\vec{H}\ \vec{x}$ may be assumed as local facts during the sub-proof of the pending obligation $A\ \vec{x}$.

This goal representation works out smoothly, as long as the main conclusion is atomic. Isabelle provides special provisions to derive non-atomic rule statements, which is treated as an "advanced method" in the Isabelle documentation [Paulson, 2001a] (here the system essentially decomposes the initial statement into an outer context and an atomic conclusion; the rule emerges implicitly by discharging the context again after finishing the proof).

The deeper reason for this inconvenience is the "improper list" representation of the outer goal structure according to HHF, which would misinterpret a non-atomic conclusion "$\ldots \Longrightarrow H$" as if the premises of H would be separate subgoals. In order to admit the rightmost position to hold arbitrary HHF formulas as well, we need to preserve the initial structure somehow. For our purposes of Isar proof composition (see chapter 3) we introduce additional proposition markers "#" (without any logical meaning) that formally turn a general "H" expression into an "A" one. *Marked HHF formulas* are of the form $G^* \Longrightarrow G$, where $G = H \mid \#H$. Only the topmost implication structure may carry markers; the outer parameter prefix is again omitted. The following derived rules admit to initialize and conclude a goal configuration (see also §3.2.3).

$$\frac{}{\varphi \Longrightarrow \#\varphi}\ (\mathit{init}) \qquad \frac{\#\varphi}{\varphi}\ (\mathit{conclude})$$

Here we only require a marker for the main conclusion. Optional markers encountered in rule premises shall play a second role to achieve proper treatment of general HHF assumptions in local goal refinements (see *refine* in §2.4.2).

2.4.2 Fundamental inference rules

Higher-order resolution composes rules via "back-chaining", while taking care of local $\bigwedge/\Longrightarrow$ contexts and instantiations automatically. Raw composition turns $\vdash \vec{A} \Longrightarrow B$ and $\vdash B \Longrightarrow C$ into $\vdash \vec{A} \Longrightarrow C$, essentially performing *modus ponens* (in reverse order), while passing through an implication prefix $\vec{A}$. The *compose* rule given below also covers implicit instantiation of the conclusion of the first rule and the premise of the second one.

$$\frac{\vec{A} \Longrightarrow B \quad B' \Longrightarrow C \quad B\,\theta = B'\theta}{\vec{A}\,\theta \Longrightarrow C\,\theta}\ (\mathit{compose})$$

Here θ shall refer to a substitution that exclusively operates on schematic variables (of types and terms, see also §2.2.1). A real implementation would typically enumerate possible solutions for θ by higher-order unification [Paulson, 1989] [Paulson, 1990], but the exact operational details do not matter here.

Actual resolution is similar to *compose*, but observes the HHF structure of the premise of the second rule. Instead of B' above the general structure may now be $\bigwedge \vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow B'\ \vec{x}$. In order admit back-chaining as indicated before, the

first rule needs to be adapted accordingly, which is called "lifting" in Isabelle jargon [Paulson and Nipkow, 1994]. Lifting over a context $\bigwedge\vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow \ldots$ may be performed by the following (derived) rules. The $\bigwedge$-*lift* rule is particularly subtle, since all schematic variables $?\vec{a}$ of the original rule need to be adapted to depend on the new outer parameters.

$$\frac{\vec{A}\ ?\vec{a} \Longrightarrow B\ ?\vec{a}}{(\bigwedge\vec{x}.\ \vec{A}\ (?\vec{a}\ \vec{x})) \Longrightarrow (\bigwedge\vec{x}.\ B\ (?\vec{a}\ \vec{x}))}\ (\bigwedge\text{-}\mathit{lift})$$

$$\frac{\vec{A} \Longrightarrow B}{(\vec{H} \Longrightarrow \vec{A}) \Longrightarrow (\vec{H} \Longrightarrow B)}\ (\Longrightarrow\text{-}\mathit{lift})$$

The *resolve* scheme is now acquired from $\bigwedge$-*lift*, $\Longrightarrow$-*lift*, and *compose*.

$$\frac{\begin{array}{l}\vec{A}\ ?\vec{a} \Longrightarrow B\ ?\vec{a} \\ (\bigwedge\vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow B'\ \vec{x}) \Longrightarrow C \\ (\lambda\vec{x}.\ B\ (?\vec{a}\ \vec{x}))\,\theta = B'\theta\end{array}}{(\bigwedge\vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow \vec{A}\ (?\vec{a}\ \vec{x}))\,\theta \Longrightarrow C\,\theta}\ (\mathit{resolve})$$

We usually prefer to write *resolve a r* in applicative order as $r \cdot a$, which may be pronounced as "r of a" (see the related operation "*OF*" introduced in §3.3.2). Resolution may be easily generalized to several argument rules $\vec{a}$ applied in parallel to a single rule r, covering a certain prefix of premises of r.

Proof-by-assumption solves a subgoal by projecting a local premise (after instantiation). Note that this may only take atomic assumptions into account, since the conclusion is atomic as well.

$$\frac{(\bigwedge\vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow A\ \vec{x}) \Longrightarrow C \quad A\,\theta = H_i\,\theta\ \ (\text{for some } i)}{C\,\theta}\ (\mathit{by\text{-}assumption})$$

Isar goal refinements essentially work just by plain resolution; the subsequent version allows arbitrary HHF assumptions to be solved at the same time. Below the first argument $\vec{G}\ ?\vec{a} \Longrightarrow B\ ?\vec{a}$ represents a local conclusion that has just been exported from a context of additional assumptions; any (optional) markers in the premises indicate immediate proof-by-assumption. The second argument $(\bigwedge\vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow B'\ \vec{x}) \Longrightarrow C$ represents an enclosing goal state with first subgoal $\bigwedge\vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow B'\ \vec{x}$; further subgoals and the conclusion are subsumed by C.

$$\frac{\begin{array}{l}\vec{G}\ ?\vec{a} \Longrightarrow B\ ?\vec{a} \\ (\bigwedge\vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow B'\ \vec{x}) \Longrightarrow C \\ (\lambda\vec{x}.\ B\ (?\vec{a}\ \vec{x}))\,\theta = B'\theta \\ (\lambda\vec{x}.\ G_j\ (?\vec{a}\ \vec{x}))\,\theta = \#H_i\,\theta\ \ (\text{for all marked } G_j \text{ for some } i)\end{array}}{(\bigwedge\vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow \vec{G}'\ (?\vec{a}\ \vec{x}))\,\theta \Longrightarrow C\,\theta}\ (\mathit{refine})$$

Recall that $\vec{G}$ may consist of marked and unmarked HHF formulas (§2.4.1). The marked ones are immediately solved against existing premises H_i, while the result $\vec{G}'$ merely covers the remaining non-marked premises of $\vec{G}$ (which become new subgoals in the result).

The *refine* operation will be hidden in the very core of the Isar proof processor (see also §3.2.3); it enables arbitrarily structured assumptions and conclusions in proof texts. Note that direct goal transformations by users (e.g. via existing tactics) never encounter the subtleties of marked versus unmarked propositions, but work with plain *resolve* or *by-assumption* steps (see also §3.3.2).

2.5 The Isabelle/Pure framework

The logical framework introduced so far may be understood as a reformed presentation of the existing Isabelle/Pure environment [Paulson, 1989] [Paulson, 1990] [Paulson and Nipkow, 1994], which will serve as the formal background for the Isar concepts introduced later on (see chapter 3). The actual Isabelle/Isar implementation [Wenzel, 2001a] has been built around the Isabelle/Pure system, too. Subsequently, we briefly review the main differences of our framework of minimal higher-order logic (§2.2) to traditional Isabelle/Pure.

The original view of higher-order logic in Isabelle/Pure [Paulson, 1989] [Paulson, 1990] is somewhat closer to older formulations [Church, 1940] [Andrews, 1986] [Gordon and Melham, 1993] [Pitts, 1993], while omitting any classical principles and choice operators, of course (see also chapter 8). The following rules have been stated by Paulson, and implemented as primitive theorem constructors in Isabelle [Paulson and Nipkow, 1994].

$$\frac{\begin{array}{c}[\varphi]\\ \vdots\\ \psi\end{array}}{\varphi \Longrightarrow \psi} \qquad \frac{\varphi \Longrightarrow \psi \quad \varphi}{\psi} \qquad \frac{\begin{array}{c}[x]\\ \vdots\\ \varphi\end{array}}{\bigwedge x.\ \varphi} \qquad \frac{\bigwedge x.\ \varphi}{\varphi[t/x]}$$

$$\frac{}{t \equiv t} \qquad \frac{t \equiv u \quad u \equiv v}{t \equiv v} \qquad \frac{t \equiv u}{u \equiv t}$$

$$\frac{f \equiv g \quad t \equiv u}{f\ t \equiv g\ u} \qquad \frac{\begin{array}{c}[x]\\ \vdots\\ t \equiv u\end{array}}{\lambda x.\ t \equiv \lambda x.\ u} \qquad \frac{}{(\lambda x.\ t)\ u \equiv t[u/x]} \qquad \frac{\begin{array}{c}[x]\\ \vdots\\ f\ x \equiv g\ x\end{array}}{f \equiv g}$$

$$\frac{\begin{array}{cc}[\varphi] & [\psi]\\ \vdots & \vdots\\ \psi & \varphi\end{array}}{\varphi \equiv \psi} \qquad \frac{\varphi \equiv \psi \quad \varphi}{\psi}$$

Here introduction and elimination of $\bigwedge/\Longrightarrow$ is the same as before (§2.2).

Definitional equality is characterized by low-level rules as an equivalence, with congruence properties wrt. λ-term formation, β-conversion, and extensionality; the correspondence to logical equivalence is expressed via explicit rules as well. In contrast, our presentation of extensional equality in §2.3 merely adds a few basic axioms to the existing framework. No rules are added, although these may be easily derived (e.g. see chapter 8). Our more compact treatment considerably simplifies meta-level studies of definitional extensions (§2.3), although the details have not been shown here.

Moreover, type instantiation is included as another primitive in Isabelle/Pure [Paulson and Nipkow, 1994]. In contrast, we have been able to acquire the same rule as an admissible one (§2.2.2), essentially due to the initial closure of axioms by type instantiations. Thus we have kept schematic polymorphism out of the core inference system (§2.2.2). Treating type instantiation as a primitive rule causes many technical subtleties of the resulting structure of derivations, much unnecessary effort has been required for the original meta-theory of overloaded definitions [Wenzel, 1994] [Wenzel, 1997].

Note that the additional concept of order-sorted type classes of Isabelle/Pure [Nipkow, 1993] [Nipkow and Prehofer, 1993] has been treated as an admissible extension of the basic inference system before [Wenzel, 1997]; see also §7.2.4 for the end-user view of type classes and overloading.

Generally speaking, our presentation of the "pure" framework has been made more conforming to common presentations of natural deduction proof systems according to typed λ-calculus. In fact, our formulation resembles the presentation of "λHOL" within Pure Type Systems [Barendregt and Geuvers, 2001]. λHOL consists of three layers of typed λ-calculus, with separate abstractions, applications, and (potentially dependent) arrow types. In our notation (§2.2) the arrows for the three layers are written as $\Rightarrow/\bigwedge/\Longrightarrow$, corresponding to syntactic function types, universal quantification, and implication, respectively. Only $\bigwedge$ may actually depend on its abstraction argument (this is an inherent property of λHOL [Barendregt and Geuvers, 2001]). The corresponding introduction and elimination rules of $\Rightarrow/\bigwedge/\Longrightarrow$ are essentially those of simple type assignment (§2.2.1), modified to operate with local typing contexts, and the basic logical inferences of higher-order natural deduction (§2.2.2).

$$\frac{\begin{array}{c}[x\colon \tau]\\ \vdots\\ t\colon \sigma\end{array}}{(\lambda x\colon \tau.\ t)\colon \tau \Rightarrow \sigma}\ (\Rightarrow\text{-}intro) \qquad \frac{t_1\colon \tau \Rightarrow \sigma \quad t_2\colon \tau}{(t_1\ t_2)\colon \sigma}\ (\Rightarrow\text{-}elim)$$

$$\frac{\begin{array}{c}[x\colon \tau]\\ \vdots\\ p\colon \varphi\end{array}}{(\lambda x\colon \tau.\ p)\colon (\bigwedge x\colon \tau.\ \varphi)}\ (\bigwedge\text{-}intro) \qquad \frac{p\colon (\bigwedge x\colon \tau.\ \varphi) \quad t\colon \tau}{(p\ t)\colon \varphi[t/x]}\ (\bigwedge\text{-}elim)$$

$$\frac{\begin{array}{c}[h\colon \varphi] \\ \vdots \\ p\colon \psi\end{array}}{(\lambda h\colon \varphi.\ p)\colon \varphi \Longrightarrow \psi}\ (\Longrightarrow\text{-}intro) \qquad \frac{p_1\colon \varphi \Longrightarrow \psi \quad p_2\colon \varphi}{(p_1\ p_2)\colon \psi}\ (\Longrightarrow\text{-}elim)$$

This unified view on minimal higher-order logic is able to improve the general theoretical understanding of the framework considerably. It has certainly influenced our simplified presentation given before (§2.2). Note that recent improvements of the Isabelle inference kernel [Berghofer and Nipkow, 2000] follow a similar perception of multi-level λ-calculus, too.

Nevertheless, the user-experience of the "real" Isabelle/Pure system differs from λHOL in a few important details. First of all, the level of mere syntactic types ("$\Rightarrow$") is left implicit most of the time, with additional conveniences like automatic type inference and polymorphism (see also §3.4.3). Moreover, the two logical levels ("$\bigwedge$" and "$\Longrightarrow$") do not expose primitive proof terms to the user, but only propositions. Essentially, a derivation object "$\vdash \varphi$" may be read as a "theorem" or "abstracted primitive proof" interchangeably. Primitive proofs never occur in actual primary proof texts of the Isar layer (cf. §1.4).

Chapter 3

The Isar proof language

We give a detailed exposition of the Isar proof language, covering syntax and operational semantics according to the Isar/VM interpretation scheme. Only the most basic elements of high-level natural deduction proof texts are hardwired as Isar primitives, further concepts are generally introduced as derived ones on top of the core system. The proof language is embedded into a generic notion of theory specifications.

Isar proof processing essentially imposes a certain policy on a selection of primitive logical operations. In particular, Isar does not introduce yet another logical calculus, but provides a conceptually different view on existing concepts of generic natural deduction, focusing on incremental language interpretation rather than primitive inference systems.

3.1 Introduction

The Isar language provides a general framework for human-readable natural deduction proofs, see also [Wenzel, 1999] for an earlier version. The Isabelle/Isar implementation [Wenzel, 2001a] enhances the Isabelle/Pure logical framework [Paulson and Nipkow, 1994] to cover actual proof texts as well. While Isar is generally somewhat biased towards that particular infrastructure of higher-order nested natural deduction, most of the basic ideas could be transferred to other foundations of mechanized logic as well.

An important philosophical issue of the Isar approach is the primacy of a high-level formal language, with an operational semantics provided by incremental interpretation. In particular, we do *not* invent a new logical calculus and establish a number of standard meta-theoretical results. Taking the very foundations of logic for granted, we build a conceptually different layer on top. As already pointed out by our terminology, the techniques to be developed here are more

appropriately related to the field of high-level programming languages, rather than mathematical logic.

Roughly speaking, the Isar language may be divided into two separate parts, for theory and proof descriptions. The latter includes both "proper" language elements for declarative proof texts, and "improper" ones for experimentation and emulation of unstructured proof scripts.

The key to viable support for human-readable formal proof texts is the design of the proper part of the Isar proof language, which consists of 12 basic elements (see also §3.2.1): "**fix** $x :: \tau$" and "**assm** $\ll r \gg$ a: A" augment the logical context, **then** indicates forward chaining of previous facts, "**have** a: A" and "**show** a: A" claim local statements (the latter includes solving of some pending goal afterwards), "**proof** m" performs an initial proof step by applying some method, "**qed** m" concludes a (sub-)proof, "**{**", "**}**" and **next** manage block structure, "**let** $p = t$" introduces term abbreviations via higher-order matching, and "**note** $a = \vec{b}$" names reconsidered facts.

Common context elements are represented as particular instances of the generic **assm** primitive, notably **assume** for the usual kind of "strong" assumptions and **def** for local definitions (see also §3.3.1). Furthermore, there are a number of derived proof commands (see §3.3.3), most notably "**by** m_1 m_2" for proofs with an empty body, "**..**" for single-rule proofs, "**.**" for immediate proofs, **hence**/**thus** for claims with forward chaining indicated, and "**from** $\vec{a}$"/"**with** $\vec{a}$" for explicit forward chaining from (additional) facts.

A few standard abbreviations are available as well: *?thesis* for the original claim at the head of the current proof, *?this* for the latest finished statement, and "..." for its left-hand side (if available). The special name *this* refers to any fact established in the previous step (**then** happens to be the same as "**from** *this*"). Fundamental proof methods are "*this*" to resolve facts directly, "(*rule r*)" to apply a rule resolved with facts, and "−" to insert previous facts without applying any rule yet (see also §3.3.2).

The natural deduction kernel of Isar directly corresponds to the underlying logical framework (cf. §2.2). For example, a meta-level rule statement may be established as follows.

have $\bigwedge x\ y\ z.\ A \Longrightarrow B \Longrightarrow C$
proof −
  fix $x\ y\ z$
  assume $A\ B$
  show C ⟨*proof*⟩
qed

Here the basic idea is to build up an Isar proof text corresponding directly to the logical connectives, using **fix** for $\bigwedge$ and **assume**/**show** for $\Longrightarrow$. In practice such proof problems usually emerge from a different claim being refined by an initial proof method, which is used instead of "−" encountered here. See chapter 4 for further basic examples on natural deduction in Isar.

Despite being primarily focused on plain natural deduction proof descriptions, the Isar framework turns out as sufficiently flexible to support a rich environment of linguistic expressions that both readers and writers may find satisfactory as a primary representation of formal documents. See chapter 5 and chapter 6 for a systematic exploration of the resulting space of useful proof patterns.

3.2 Syntax and semantics

The syntax of the Isar theory and proof specification language is defined via a set of *primary commands*, as defined by the syntactic category *command* below (§3.2.1). From the syntactic viewpoint, any sequence of commands is already a well-formed Isar document. The semantics of each command is determined by a *transition* of the underlying configuration, which may be either the background library of theories, an individual theory, or a proof state (which is again subdivided into three different modes). From the basic typing of commands induced by the transition semantics we impose a certain structure of Isar documents, which achieves a block structured formal language that may be presented in the usual form via a context free grammar retrospectively (see §3.2.4).

By having the Isar language emerge in this bottom-up manner we emphasize its incremental interpretation, such that the process of formal proof checking coincides with that of interactive development and debugging (commands turn out as sufficiently fine-grained to support this in reality).

Moreover we may easily extend the basic language by derived commands, which are defined as abbreviations of existing ones (potentially depending on the current state). Derived elements may be freely combined with the rest of the language, according to the typing determined by the semantics of the underlying primitives. Thus we achieve a maximal degree of language compositionality for free, without having to maintain a fixed global grammar.

Consequently, the Isar primitives may be restricted to the bare minimum required to bootstrap the language environment. A number of standard derived elements introduced later on (see §3.3.3) are indispensable even for the most basic applications. Further canonical slots for extensions are that of proof methods (goal refinement schemes) and attributes (operations involving facts); the most basic ones are included in the general Isar setup (see §3.3.2).

In the subsequent definitions of syntactic categories related to the basic Isar language, we are using the following notation for regular expressions, namely parentheses "(...)" for grouping, x^* for zero or more occurrences of a language element x, likewise x^+ for one or more, and $x^?$ for zero or one occurrences of x. Furthermore, recall the following basic formal items introduced in §2.1 and §2.2: *nat* for natural numbers, *bool* for truth values, *name* for basic names, *type* for well-formed (simple) types, *term* for well-typed λ-terms, *prop* for propositions (terms of type *prop*), and *theorem* for derivable propositions.

3.2.1 Isar commands

The Isar proof language consists of three main layers, with these sub-languages:

command	primary proof commands
method	operations on goals
attribute	operations involving facts

Presently our main focus is on the primary command language. The secondary ones of *method* and *attribute* may be considered as a parameter of the whole Isar framework. Later on we will merely specify a few fundamental methods and attributes (see §3.3.2), while leaving it to concrete working environments to incorporate any further tools as appropriate (like the collection of automated proof methods of Isabelle/HOL, see §7.3).

The primary language *command* defines a number of primitives (both for theory and proof operations) as follows.

command =
  theory *name* = *name* (+ *name*)*:
| **end**

| **types** (*name nat*)+
| **consts** (*name* :: *type*)+
| **axioms** (*name-atts*: *prop*)+
| **theorems** (*name-atts* =)? *name-atts*+
| **theorem** (*name-atts*:)? *prop*

| **apply** *method*
| **done**

| **proof** *method*?
| **qed** *method*?
| **{**
| **}**
| **next**
| **let** *term* = *term* (**and** *term* = *term*)*
| **note** (*name-atts* =)? *name-atts*+ (**and** (*name-atts* =)? *name-atts*+)*
| **fix** *var*+
| **assm** «*rule*» (*name-atts*:)? *prop*+ (**and** (*name-atts*:)? *prop*+)*
| **then**
| **have** (*name-atts*:)? *prop*
| **show** (*name-atts*:)? *prop*

Some additional basic categories are defined below.

var	=	*name* × *type*
case	=	*var** × *prop**
fact	=	*theorem**
rule	=	*theorem* → *theorem*
name-atts	=	*name* [*attribute**]?

In order to simplify the subsequent treatment of commands, we fix the default values for any optional arguments as follows.

proof	default method *rule* (see §3.3.2)
qed	default method *succeed* (see §3.3.2)
name-atts	default name *this* with empty list of attributes

Also note that the *rule* argument of **assm** is treated in a special way, such that Isar proof texts given by users may not refer to it directly (there is no concrete syntax for *rule* available). Thus **assm** may only occur indirectly via derived commands, such as the basic context elements **assume**, **presume**, **def**, and **case** introduced in §3.3.1.

3.2.2 Basic types of commands

A formal Isar text is syntactically correct iff it conforms to the (degenerate) grammar $command^+$. Certainly, this does not yet impose any specific structure on formal texts, which will be only determined as part of the operational semantics of commands, involving fine-grained typings.

Assume the types *library* (for the background storage of theories), *theory* (for theory contexts), and *proof* (for proof configurations). Isar commands are given the following types expressed in the signature diagram of figure 3.1.

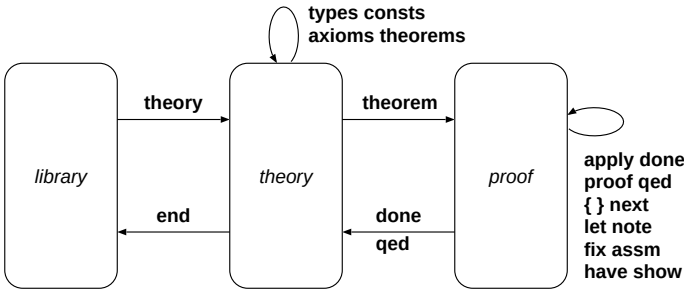

Figure 3.1: Basic types of Isar commands

Corresponding to these basic typings, we also introduce the following classification of Isar commands.

theory setup commands	**theory**, **end**
theory specifications	**types**, **consts**, **axioms**, **theorems**, **theorem**
improper proof commands	**apply**, **done**
proper proof commands	**proof**, **qed**, **{**, **}**, **next**, **let**, **note**, **fix**, **assm**, **then**, **have**, **show**

Naturally, proof commands and proof configurations are the main focus of Isar, see §3.2.3 for further details.

Presently we shall point out a few aspects of how the Isar concept of proof may be embedded into theory specifications, including integration with an enclosing library of theories. The underlying concepts of *theory* and *library* happen to be closely related to those of recent versions of Isabelle [Paulson, 2001b] [Berghofer and Wenzel, 2001]. Nevertheless, this particular model mainly serves as a working example, obviously the Isar proof language could be also embedded into rather different theory concepts as well.

A *theory* consists of purely logical declarations (according to §2.2), together with an explicit environment of facts, and an additional slot to keep any kind of auxiliary information *data*.

record *theory* =
  types :: *set of* (*name* × *nat*)
  consts :: *set of* (*name* × *type*)
  axioms :: *name* ⇀ *prop*
  theorems :: *name* ⇀ *fact*
  data :: *data*

The *data* slot is left unspecified for our present purpose. While it is kept along with the theory, it does not affect its meaning from the purely logical point of view. Nevertheless, the concept of extra-logical theory data proves an indispensable tool to support advanced theorem proving environments (e.g. for separate contexts of rules to be used with automated proof procedures, as well as high-level theory specifications; see also chapter 7).

Commands **types**, **consts**, **axioms** shall maintain the corresponding primitive theory fields in the obvious manner; **theorems** provides a direct interface to update the theorem environment; **theorem** is conceptually quite different from the previous ones as it first enters a *proof* configuration, eventually resulting in an actual theorem, and then updates *theorems* accordingly. The theorem names "-" and *nothing* shall be considered as reserved, with the standard assignments of $\vdash \bigwedge A.\ A \Longrightarrow A$ and the empty fact, respectively.

A *library* shall represent any kind of background storage of individual *theory* objects, usually with some inherent notion of (acyclic) dependencies.

record *library* =
  theories :: *name* ⇀ *theory*
  deps :: *name* ⇀ *set of name*

Viable theory management for large-scale applications is still an issue of ongoing research, both from the logical perspective (e.g. [Pollack, 2000]) and the change management view (e.g. [Reif, 1992] [Hutter, 2000]). Concerning Isar we only demand the primitive "**theory** $a = b_1 + \ldots + b_n$:" for commencing a new theory context from the merge of existing ones, and **end** to put the result back into the library. Independently of any automatic mechanisms of update,

change management, synchronization with external repositories etc., the core Isar commands operate on *theory* and *proof* configurations in a linear fashion. We deliberately rule out unstructured interaction with the theory arrangement once that a particular context has been entered. For example, there is no command for ad-hoc *import* of existing theories into the present context. The idea is to provide commands to compose a collection of well-defined theory documents, rather than ad-hoc manipulations of formal entities.

3.2.3 Isar/VM transitions

We are ready to define the operational semantics of the actual Isar proof commands, by interpretation as transitions of the *Isar virtual machine* (*Isar/VM*).

According to definitions to be given later on, Isar/VM configurations have type *proof*, mainly consisting of a static proof *context* plus dynamic *goal* information. $\mathcal{C}$ shall fix the semantics of proof commands (together with initial and terminal linking with the theory context), while interpretations $\mathcal{M}$ for proof methods (with and without additional case bindings) and $\mathcal{A}$ for attributes (both for proof and theory contexts) are left as parameters.

$$
\begin{array}{lcl}
\mathcal{C} & : & \mathit{command} \to \mathit{theory} \to \mathit{proof} \\
\mathcal{C} & : & \mathit{command} \to \mathit{proof} \to^{**} \mathit{proof} \\
\mathcal{C} & : & \mathit{command} \to \mathit{proof} \to \mathit{theory} \\
\mathcal{M} & : & \mathit{method} \to \mathit{context} \to \mathit{fact} \to \mathit{tactic} \times (\mathit{name} \rightharpoonup \mathit{case}) \\
\mathcal{M} & : & \mathit{method} \to \mathit{context} \to \mathit{fact} \to \mathit{tactic} \\
\mathcal{A} & : & \mathit{attribute} \to \mathit{context} \times \mathit{theorem} \to \mathit{context} \times \mathit{theorem} \\
\mathcal{A} & : & \mathit{attribute} \to \mathit{theory} \times \mathit{theorem} \to \mathit{theory} \times \mathit{theorem}
\end{array}
$$

A *proof* configuration is defined as a stack of basic proof states: $\mathit{proof} = \mathit{state}^*$. The stack represents the block structure of the proof text; proof commands usually operate on the head of the stack only, except those that affect the block structure itself.

record *state* =

mode	::	prove | state | chain
context	::	*context*
goal	::	*goal* | none

Modes of operation

We distinguish three fundamental modes of operations of the Isar/VM, with the following informal meaning:

prove	awaiting direct transformation of the present claim (by method application)
state	ready to state new local items (assumptions, local claims etc.)
chain	awaiting a new claim, with previous facts being indicated for later use

Isar proof commands acquire certain typings according to these three modes, as shown in figure 3.2. This imposes an inherent structure on Isar proof texts, according the role that proof commands may play in a particular situation. Only those sequences of proof commands may get successfully processed by the Isar/VM interpreter that correspond to legal paths of this diagram. Note that further structural constraints are achieved via proper nesting of blocks, which is not encoded into the *mode* field, but determined from the stack structure.

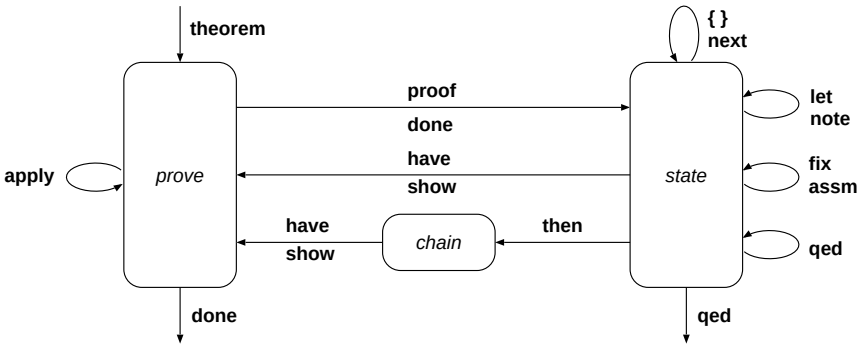

Figure 3.2: Transitions of Isar proof processing

This inherent fine-grained typing of proof states is a key concept of Isar/VM proof processing. Thus we achieve both well-structured texts and incremental checking of individual commands. In contrast, traditional tactic scripts would operate only on a single kind of state, as may be even expressed within the Isar framework. By restriction to improper proof commands the original Isar/VM diagram of figure 3.2 degenerates to that of figure 3.3.

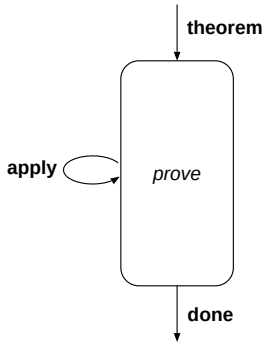

Figure 3.3: Transitions of tactical theorem proving

Apparently, tactical theorem proving is like holding your breath, until the present claim has been solved completely by direct goal refinements via meth-

ods. Proper Isar proof texts would usually only perform one or zero direct steps, in order to change quickly into the rich landscape of `state` mode in order to explore the present context in a casual manner. See §4.2.3 for further discussion of the issue of "operational" versus "declarative" theorem proving in practice.

State components

Apart from the *mode* field, Isar proof states have clearly separated components of "static" proof *context* versus "dynamic" *goal* information. As a general principle, the context keeps all those items that directly correspond to declarations given in the text (assumptions, finished claims, term and fact bindings etc.), while the goal state contains the leftover claim that may have undergone several direct refinements beforehand. In the operational semantics we take care that results of proof methods may never intrude the subsequent proof text.

record *context* =
  theory :: *theory*
  fixes :: var^*
  assms :: $rule^*$
  terms :: $name \rightharpoonup term$
  cases :: $name \rightharpoonup case$
  facts :: $name \rightharpoonup fact$
  data :: *data*

Here *theory* is kept for reference to the enclosing context; it will only be changed in a final proof step, just before handing over back to the theory level.

The *fixes* and *assms* fields correspond to logical contexts $\bigwedge \vec{x}.\ \vec{H} \Longrightarrow \ldots$ in HHF normal form (cf. §2.4.1). Instead of plain propositions $\vec{H}$ we rather keep the corresponding discharge rules of assumptions (see also §3.3.1).

In addition, we maintain auxiliary environments of *terms*, *cases*, *facts*, which do not have any immediate logical significance, but are indispensable to support the course of reasoning in a well-structured high-level manner, as it enables proof commands to refer to bits and pieces of logical entities under construction.

The slot for arbitrary *data*, which is inherited from the theory, may hold any further information, e.g. separate contexts for dedicated proof tools (see §7.3).

record *goal* =
  solve :: *bool*
  name :: *name*
  atts :: $attribute^*$
  statement :: *prop*
  using :: *fact*
  problem :: *theorem*

The *goal* fields mostly contain bookkeeping information to capture the present state of pending claims: *solve* distinguishes **have** from **show** (the latter is

intended to refine an enclosing claim when finished); *name* and *atts* keep the original declaration, which is applied to the result eventually; *statement* holds the original claim; *using* tracks any facts indicated for forward chaining (as indicated via **then**), otherwise the empty list; *problem* holds the internal goal state, represented as a theorem of the underlying logical framework (cf. §2.4).

Incidently, the order of individual constituents of Isar proof configurations presented above roughly corresponds to their significance in practical application of the framework: the static context (which corresponds to a piece of proof text up to a certain position) is of more interest than the dynamic goal information (which is only relevant in isolated situations where direct refinement takes place). In contrast, traditional tactical proving mostly revolves around the *goal*/*problem* part, while being ignorant of the rest.

Nevertheless, the very purpose of that "redundant" apparatus of Isar/VM proof configurations is to support an interpretation model of formal proof texts that enables the writer to arrange the course of reasoning in such a way that the reader is liberated from taking into account any accidental behavior of primitive operations underlying arbitrary proof methods.

Interpreting commands

We are ready to define the interpretation $\mathcal{C}$ of proof commands as Isar/VM transitions; recall that $\mathcal{M}$ and $\mathcal{A}$ are still left open. Refer to chapter 2 for basic operations, such as record field manipulations, composition of theorems (rules) of the logical framework etc.

In order to avoid excessive notational detail, we use the convention that record operations may be also applied to more complex structures (such as the stack over *state*, or the record *state* over *context* and *goal*). Furthermore, partial functions and procedures (cf. §2.1) shall be converted implicitly as required, so plain functional notation may be used throughout our specification. For brevity the main definition of $\mathcal{C}$ is presented in combinatorial style, without ever mentioning the configuration to be transformed explicitly. Recall procedural composition p; q and alternative choice $p \mid q$ from §2.1.

$\mathcal{C}$ (**theorem** q: φ) = *init-proof*; *open-block*; *prepare-term* φ; *bind-goal*; *init-goal* false q; *put-mode* prove
$\mathcal{C}$ (**apply** m) = *assert-mode* {prove}; *transform-goal* m; *put-using* []
$\mathcal{C}$ (**done**) = *assert-mode* {prove}; *conclude-goal*; *refine-enclosing*; (*store-result* | (*bind-result*; *set-this*; *put-mode* state))
$\mathcal{C}$ (**proof** m) = *assert-mode* {prove}; *transform-goal* m; *put-mode* state
$\mathcal{C}$ (**qed** m) = *assert-mode* {state}; *assert-goal* true; *transform-goal* m; *transform-goal finish*; *conclude-goal*; *refine-enclosing*; (*store-result* | (*bind-result*; *set-this*; *put-mode* state))

$\mathcal{C}$ (**{**) = *assert-mode* {state}; *open-block*; *put-goal* none; *open-block*
$\mathcal{C}$ (**}**) = *assert-mode* {state};
export-this (*close-block*; *assert-goal* false; *close-block*); *set-this*
$\mathcal{C}$ (**next**) = *assert-mode* {state}; *close-block*; *open-block*;
reset-this; *put-mode* state
$\mathcal{C}$ (**let** $p_1 = t_1$ **and** ... **and** $p_n = t_n$) = *assert-mode* {state};
prepare-terms [t_1, ..., t_n]; *bind-terms* [p_1, ..., p_n]; *reset-this*
$\mathcal{C}$ (**note** $q_1 = \vec{r}_1$ **and** ... **and** $q_n = \vec{r}_n$) = *assert-mode* {state};
prepare-facts [$\vec{r}_1$, ..., $\vec{r}_n$]; *bind-facts* [q_1, ..., q_n]; *set-this*
$\mathcal{C}$ (**fix** $\vec{x}$) = *assert-mode* {state}; *map-fixes* (λ*fixes*. *fixes* @ $\vec{x}$); *reset-this*
$\mathcal{C}$ (**assm** $\ll r \gg$ q_1: $\vec{\varphi}_1$ **and** ... **and** q_n: $\vec{\varphi}_n$) = *assert-mode* {state};
prepare-termss [$\vec{\varphi}_1$, ..., $\vec{\varphi}_n$]; *add-assms* r; *bind-facts* [q_1, ..., q_n]; *set-this*
$\mathcal{C}$ (**then**) = *assert-mode* {state}; *put-mode* chain
$\mathcal{C}$ (**have** q: φ) = *assert-mode* {state, chain}; *open-block*;
prepare-term φ; *bind-goal*; *init-goal* false q; *put-mode* prove
$\mathcal{C}$ (**show** q: φ) = *assert-mode* {state, chain}; *open-block*;
prepare-term φ; *bind-goal*; *init-goal* true q; *put-mode* prove

The special proof method *finish* encountered above ensures that a goal configuration $\vdash \vec{\chi} \Longrightarrow \#\varphi$ is reduced to $\vdash \#\varphi$ with all remaining subgoals solved *by-assumption* (§2.4.2), either directly or after applying a single rule from the current *prems* via *resolve* (§2.4.2).

Several auxiliary functions for $\mathcal{C}$ still need to be defined (see below). This will ultimately provide a mathematical model of the Isar/VM interpretation process of Isabelle/Isar [Wenzel, 2001a]. The above presentation of $\mathcal{C}$ may already serve as a semi-formal exposition of the general idea of Isar proof processing.

Internal operations

For the subsequent definitions of internal operations encountered in $\mathcal{C}$ above, we use the notational conventions of Θ for a *theory* argument, ϑ for *theorem*, $\vec{\vartheta}$ for *fact*, σ for *state*, and $\vec{\sigma}$ for *proof*. Also recall backwards application $x \triangleright f$ (binding tightly) from §2.1.

A few names of the Isar language shall be reserved, in the sense that these may never be bound directly in proof texts by users, but only internally by (primitive or derived) commands. These are *?thesis*, *?this*, "..." for terms, *this*, *prems* for facts, and *antecedent* for cases.

An initial proof configuration merely consists of an initial context, which in turn contains the enclosing *theory* (which is never changed until the very end of the main proof), and inherits the global theorem environment and auxiliary *data*.

init-proof Θ = [(|*mode* = state, *context* = *init-context* Θ, *goal* = none|)]
init-context Θ = (|*theory* = Θ, *fixes* = [], *assms* = [], *terms* = {},
cases = {}, *facts* = (*get-theorems* Θ)(*prems* := []), *data* = *get-data* Θ|)

Basic block management is mediated by the inherent stack structure of proof configurations.

open-block $(\sigma \circ \vec{\sigma}) = \sigma \circ \sigma \circ \vec{\sigma}$
close-block $(\sigma \circ \vec{\sigma}) = \vec{\sigma}$

The following assertions ensure certain types of proof configurations encountered during operation of the Isar/VM, the interpretation process simple stops on failure of any such condition. Note that a *goal* record is always kept behind an additional level of nesting, which enables **next** to jump blocks as expected.

assert-mode $M\ \sigma =$
  if *get-mode* $\sigma \in M$ then σ else *undefined*
assert-goal $b\ (\sigma_1 \circ \sigma_2 \circ \vec{\sigma}) =$
  if $b = (\textit{get-goal}\ \sigma_2 \neq \text{none})$ then $\sigma_1 \circ \sigma_2 \circ \vec{\sigma}$ else *undefined*

The special fact binding *this* is maintained to hold the most recent result, otherwise it is undefined in order to cause failure on forward-chaining out of nothing.

set-this $(\sigma, \vec{\vartheta}) = \sigma \triangleright \textit{map-facts}\ (\lambda e.\ e(\textit{this} := \vec{\vartheta}))$
reset-this $= \textit{map-facts}\ (\lambda e.\ e(\textit{this} := \textit{undefined}))$

The *export* operation performs fundamental adjustments required to move a theorem out of a context of local parameters and assumptions, essentially the context difference is discharged inside-out. The $\bigwedge$*-intro* rule is from §2.2.2 and *generalize* from §2.4.

export $\sigma\ \sigma'\ \vartheta =$
  let $[x_1, \ldots, x_n] = \textit{get-fixes}\ \sigma - \textit{get-fixes}\ \sigma'$;
    $[r_1, \ldots, r_n] = \textit{get-assms}\ \sigma - \textit{get-assms}\ \sigma'$
  in $\vartheta \triangleright r_n \cdots \triangleright r_1 \triangleright \bigwedge\textit{-intro}\ x_n \triangleright \cdots \triangleright \bigwedge\textit{-intro}\ x_1 \triangleright \textit{generalize}\ \vec{x}$

Terms entered into the text are always normalized with respect to the current environment of term bindings. The basic operation *norm e t* shall replace all occurrences of schematic variables *?x* by the term *norm e* (*e ?x*), i.e. lookup the binding and normalize recursively; this operation fails for unbound variables. Moreover, *unify* shall perform simultaneous higher-order unification on pairs of terms, enumerating possible result bindings. Note that immediate term bindings are covered by degenerate patterns consisting of a single variable only. The special dummy pattern "-" refers to a "fresh" schematic variable for each occurrence; this allows to specify patterns where certain positions are skipped.

prepare-term $t\ \sigma = (\sigma, \textit{norm}\ (\textit{get-terms}\ \sigma)\ t)$
prepare-terms $ts\ \sigma = (\sigma, \textit{map}\ (\textit{norm}\ (\textit{get-terms}\ \sigma))\ ts)$
prepare-termss $tss\ \sigma = (\sigma, \textit{map}^2\ (\textit{norm}\ (\textit{get-terms}\ \sigma))\ tss)$
bind-terms $[p_1, \ldots, p_n]\ (\sigma, [t_1, \ldots, t_n]) =$
  $\sigma \triangleright \textit{map-terms}\ (\lambda e.\ e +\!\!+ \textit{unify}\ [(p_1, t_1), \ldots, (p_n, t_n)])$

Explicit statements in the text give rise to automatic bindings of reserved names, depending on the present role as a goal or result statement. It is important to

note that we never peek at theorems (neither facts nor goals), but merely analyze terms stemming from the text. A proposition φ of the form $\bigwedge \vec{x}.\ \vec{H}\ \vec{x} \Longrightarrow A\ \vec{x}$ is decomposed as follows: *conclusion-of* φ yields the term A, and *antecedent-of* φ yields the case $(\vec{x},\ \vec{H})$, and *argument-of* φ yields the right-hand side of A if that is an application (otherwise *undefined*).

bind-goal $(\sigma,\ \varphi) = (\sigma$
  $\triangleright$ *map-terms* $(\lambda e.\ e(?thesis := conclusion\text{-}of\ \varphi))$
  $\triangleright$ *map-cases* $(\lambda e.\ e(antecedent := antecedent\text{-}of\ \varphi)),\ \varphi)$
bind-statement $\varphi\ \sigma = \sigma$
  $\triangleright$ *map-terms* $(\lambda e.\ e(?this := conclusion\text{-}of\ \varphi)(\text{“}\ldots\text{”} := argument\text{-}of\ \varphi))$

The goal operations given below manage the dynamic component of a proof configuration. Initially, we setup a *goal* record consisting of the result specification, the presently chained facts, and a trivial proof state represented as a theorem. The *init* rule is from §2.4.

init-goal solve $(name,\ atts)\ (\sigma,\ \varphi) =$
  $\sigma \triangleright$ *put-goal* $(\!|solve = solve,\ name = name,\ atts = atts,\ statement = \varphi,$
    $using =$ if *get-mode* $\sigma =$ chain then *get-facts* σ *this* else $[],$
    $problem = init\ \varphi|\!) \triangleright$ *reset-this* $\triangleright$ *open-block* $\triangleright$ *put-goal* none

Goal transformations are the only occasion for application of proof methods, which may refer to arbitrary tactics inside (cf. §2.4).

transform-goal $m\ (\sigma_1 \circ \sigma_2 \circ \vec{\sigma}) =$
  let $(tactic,\ cases) = \mathcal{M}\ m\ (get\text{-}context\ \sigma_1)\ (get\text{-}using\ \sigma_2);$
    $add\text{-}cases = map\text{-}cases\ (\lambda e.\ e +\!\!+ cases)$
  in $(\sigma_1 \triangleright add\text{-}cases) \circ (\sigma_2 \triangleright map\text{-}problem\ tactic \triangleright add\text{-}cases) \circ \vec{\sigma}$

The operation of concluding a goal exhibits the finished result in two ways, both for export into the enclosing goal (if applicable), and for immediate binding in the present context. The *generalize* and *conclude* rules are from §2.4.

conclude-goal $(\sigma_1 \circ \sigma_2 \circ \vec{\sigma}) =$
  let $goal = get\text{-}goal\ \sigma_2;$
    $\vartheta = generalize\ (conclude\ (get\text{-}problem\ goal));$
    $result = (get\text{-}name\ goal,\ get\text{-}atts\ goal,\ get\text{-}statement\ goal,\ \vartheta)$
  in $((\vec{\sigma},\ (get\text{-}solve\ goal,\ export\ \sigma_2,\ \vartheta)),\ result)$

In order to refine the enclosing problem (the innermost according to the structure of sub-proofs), we search the stack of proof states upwards and apply the given theorem after export with respect to the context difference. The all-important *refine* rule encountered here is from §2.4.2, which takes care of both the conclusion and assumptions of a subgoal (according to the “#” markers attached to premises after export). The *select* operation shall traverse subgoals from left-to-right, enabling the subsequent *refine* to succeed on the first match.

refine-enclosing $(\vec{\sigma}, (\mathit{solve}, \mathit{exp}, \vartheta)) =$
  if $\neg$ *solve* then $\vec{\sigma}$
  else $\vec{\sigma} \triangleright$ *map-enclosing* $(\lambda\sigma'.\ \mathit{map\text{-}problem}\ (\mathit{refine}\ (\mathit{exp}\ \sigma'\ \vartheta) \circ \mathit{select}))$
map-enclosing $f\ (\sigma \circ \vec{\sigma}) =$
  if *get-goal* $\sigma \neq$ none then *map-goal* $(f\ \sigma) \circ \vec{\sigma}$ else $\sigma \circ$ *map-enclosing* $f\ \vec{\sigma}$

Export of facts without a goal context is covered below. The *purge* operation shall remove any "#" markers (§2.4) that may have got introduced by *export* (markers are only significant for actual goal refinements).

export-this outer $\vec{\sigma} = (\vec{\sigma}, \mathit{purge}\ (\mathit{export}\ \vec{\sigma}\ (\mathit{outer}\ \vec{\sigma})\ (\mathit{get\text{-}facts}\ \vec{\sigma}\ \mathit{this})))$

Any facts emerging in the proof text (assumptions, finished goals etc.) may be modified by attribute expressions. Recall that $\mathcal{A}$ interprets attributes as *context* $\times$ *theorem* $\to$ *context* $\times$ *theorem* or *theory* $\times$ *theorem* $\to$ *theory* $\times$ *theorem*, depending on the context. $\mathcal{A}$ is lifted to lists of attributes via sequential composition (left-to-right): $\mathcal{A}^*\ [\alpha_1, \ldots, \alpha_m] = \mathcal{A}\ \alpha_1; \ldots; \mathcal{A}\ \alpha_m$. The *apply-facts* function given below evaluates lists of pairs of facts and attributes, returning the modified context and accumulated results. For a single theorem this works as follows: *apply-facts* $(c, [([\vartheta], \vec{\alpha})]) =$ let $(c', \vartheta') = \mathcal{A}^*\ \vec{\alpha}\ (c, \vartheta)$ in $(c', [[\vartheta']])$. The general definition uses the *iterate* combinator from §2.1.

apply-facts $=$ *iterate* $(\lambda(c, (\vec{\vartheta}, \vec{\alpha}))$. *iterate* $(\mathcal{A}^*\ \vec{\alpha})\ (c, \vec{\vartheta}))$

Referenced facts are retrieved from the environment and modified by attributes. Binding of facts may involve additional attributes on the left-hand side of the specification, which are applied just before the actual environment update.

prepare-facts $[(a_1, \vec{\alpha}_1), \ldots, (a_n, \vec{\alpha}_n)]\ \sigma =$
  apply-facts $(\sigma, [(\mathit{get\text{-}facts}\ \sigma\ a_1, \vec{\alpha}_1), \ldots, (\mathit{get\text{-}facts}\ \sigma\ a_n, \vec{\alpha}_n)])$
bind-facts $[(a_1, \vec{\alpha}_1), \ldots, (a_n, \vec{\alpha}_n)]\ (\sigma, [\vec{\vartheta}_1, \ldots, \vec{\vartheta}_n]) =$
  let $(\sigma', [\vec{\zeta}_1, \ldots, \vec{\zeta}_n]) =$ *apply-facts* $(\sigma, [(\vec{\vartheta}_1, \vec{\alpha}_1), \ldots, (\vec{\vartheta}_n, \vec{\alpha}_n)])$
  in $(\sigma' \triangleright$ *map-facts* $(\lambda e.\ e +\!\!+ [(a_1, \vec{\zeta}_1), \ldots, (a_n, \vec{\zeta}_n)]), \vec{\zeta}_1 \mathbin{@} \ldots \mathbin{@} \vec{\zeta}_n)$

The proven result of a finished proof is either put back into the enclosing theory (concluding the main proof altogether), or bound in the local proof context for continued proof operation.

store-result $([\sigma], (a, \vec{\alpha}, \varphi, \vartheta)) =$
  let $\Theta =$ *get-theory* σ; $(\Theta', [[\vartheta']]) =$ *apply-facts* $(\Theta, [([\vartheta], \vec{\alpha})])$
  in $\Theta' \triangleright$ *map-theorems* $(\lambda e.\ e(a := [\vartheta']))$
bind-result $(\sigma, (a, \vec{\alpha}, \varphi, \vartheta)) =$
  $(\sigma \triangleright$ *bind-statement* $\varphi, [[\vartheta]]) \triangleright$ *bind-facts* $[(a, \vec{\alpha})]$

Assumptions are introduced in chunks, giving a list of proposition lists. This additional structure merely serves for separate naming of the resulting local facts, premises are flattened internally. The *assumption* rule is from §2.2.2 and *generalize* from §2.4.

$$\begin{array}{l} \mathit{add\text{-}assms}\ r\ (\sigma, [\vec{\varphi}_1, \ldots, \vec{\varphi}_n]) = \\ \quad \mathsf{let}\ [\vec{\vartheta}_1, \ldots \vec{\vartheta}_n] = \mathit{map}^2\ (\mathit{generalize} \circ \mathit{assumption})\ [\vec{\varphi}_1, \ldots, \vec{\varphi}_n] \\ \quad \mathsf{in}\ (\sigma \triangleright \mathit{map\text{-}assms}\ (\lambda \mathit{assms}.\ \mathit{assms}\ @\ [r]) \triangleright \mathit{bind\text{-}statement}\ (\mathit{last}\ \vec{\varphi}_n) \\ \qquad \triangleright \mathit{map\text{-}facts}\ (\lambda e.\ e(\mathit{prems} := e\ \mathit{prems}\ @\ \vec{\vartheta}_1\ @\ \ldots\ @\ \vec{\vartheta}_n)), [\vec{\vartheta}_1, \ldots \vec{\vartheta}_n]) \end{array}$$

An example interpretation

We briefly review the operation of the Isar/VM interpreter on a small synthetic example, which particularly illustrates the policy of interpretation that eventually leads to application of primitive inferences (notably *refine* given in §2.4.2). The proof text fragment below merely covers the main elements of generic **assm** and **show** in the context of a different goal (**have**). In fact, most other Isar commands mostly perform bookkeeping steps only, which essentially serve as a preparation for such fundamental incidents of actual goal refinement.

1. **have** $A \Longrightarrow B$
2. **proof** *succeed*
3. **assm** $\ll \mathit{disch} \gg A$
4. **show** B
5. $\langle \mathit{proof} \rangle$
6. **qed**

$$\frac{\Gamma \cup \{A\} \vdash \varphi}{\Gamma \vdash \#A \Longrightarrow \varphi}\ (\mathit{disch})$$

Here we are still confined to certain "raw" expressions of Isar commands that would normally not occur in reality, i.e. the identity method *succeed* (see §3.3.2) and the **assm** primitive supplied with an inference rules for discharging (and marking) the assumption. See also §3.3 for definitions of derived commands and proof methods for actual end-user applications.

Subsequently, we give the generated sequence of internal operations, while suppressing successful assertions (which merely result in identities). The second column below exhibits the trace of primitive inferences encountered during interpretation. The initial proof configuration shall be in state mode. We further assume that "$\langle \mathit{proof} \rangle$" refers to a successful sequence of commands ending with **done** or **qed** (returning to the original nesting level of the corresponding **show**).

1. $\mathcal{C}$ (**have** $A \Longrightarrow B$):

   open-block
   prepare-term $(A \Longrightarrow B)$
   bind-goal
   init-goal false (*this*, []) $\quad \mathit{init}\ (A \Longrightarrow B) = \vdash (A \Longrightarrow B) \Longrightarrow \#(A \Longrightarrow B)$
   put-mode prove

2. $\mathcal{C}$ (**proof** *succeed*):

   transform-goal succeed
   put-mode state

3. $\mathcal{C}$ (**assm** $\ll$*disch*$\gg$ A):

prepare-termss $[[A]]$	
add-assms disch	*assumption* $A = \{A\} \vdash A$
bind-facts $[(this, [])]$	
set-this	

4. $\mathcal{C}$ (**show** B):

open-block	
prepare-term B	
bind-goal	
init-goal true $(this, [])$	*init* $B = \vdash B \Longrightarrow \#B$
put-mode prove	

5. $\mathcal{C}$ $\langle$*proof*$\rangle$:

$\vdots$	
conclude-goal	*conclude* $(\{A\} \vdash \#B) = \{A\} \vdash B$
refine-enclosing	*disch* $(\{A\} \vdash B) = \vdash \#A \Longrightarrow B$
	refine $(\vdash \#A \Longrightarrow B)$
	$(\vdash (A \Longrightarrow B) \Longrightarrow \#(A \Longrightarrow B)) = \vdash \#(A \Longrightarrow B)$
bind-result	
set-this	
put-mode state	

6. $\mathcal{C}$ (**qed**):

transform-goal succeed	
transform-goal finish	
conclude-goal	*conclude* $(\vdash \#(A \Longrightarrow B)) = \vdash A \Longrightarrow B$
refine-enclosing	
bind-result	
set-this	
put-mode state	

The terminal context now holds a fact assignment of $this = [\vdash A \Longrightarrow B]$.

3.2.4 Recovering static syntax

As the most basic property of the Isar/VM transition system we shall see how "static" syntax of proof texts may be recovered, by deriving a context-free grammar that approximates the language of legal proof texts. To this end we inspect the definition of $\mathcal{C}$ (§3.2.3) from an abstract viewpoint, such that only the *mode* field and the stack structure (block nesting) is taken into account.

The only commands that may actually affect blocks are either goals (**theorem**, **have**, **show**), conclusions of proofs (**done**, **qed**), or separate block commands ("**{**", "**}**", **next**). Blocks are always opened in pairs, with an optional goal placed in between. So we may either get a "goal sandwich" of the form "$\langle$*goal*$\rangle$" as produced by goal commands, or a plain block opening "$\langle\langle$" produced by "**{**".

We also see that proof conclusions exactly match goal sandwiches and "**}**" plain openings, while **next** copes with either case (preserving it). Consequently, proof and statement blocks are always properly nested, without interfering each other.

Now consider the *mode* behavior of proof commands given in figure 3.2. Let *proof-statement* refer to the sub-language of commands that are successfully processed in `state` mode. Apparently this category consists of properly nested blocks, basic context statements, or goal statements with a complete proof (after an optional chain indicator **then**). The category *proof* is unfolded from `prove` mode in a similar fashion: it consists of (optional) initial scripts of **apply** elements, followed by properly nested proof texts, or the **done** terminator.

Together with the linking of the proof language to the theory level (as indicated in figure 3.1) we may now easily complete a syntactic approximation of well-formed Isar proof texts by giving the following grammar (below we also omit command arguments for clarity).

theory-statement	=	**types** | **consts** | **axioms** | **theorems**
	|	**theorem** *proof*
proof	=	**apply*** (*proper-proof* | **done**)
proper-proof	=	**proof** *proof-statement** **qed**
proof-statement	=	**{** *proof-statement** **}**
	|	**next** | **let** | **note** | **fix** | **assm**
	|	**then**$^?$ *goal-statement*
goal-statement	=	**have** *proof* | **show** *proof*

This grammar could be refined further, e.g. to include the state of current facts as well. Consequently, legal use of **then** could be specified more explicitly by grammatical means, ruling out malformed phrases like "**next then**".

Note that the above presentation only covers the core language of Isar, stemming from the basic set of commands considered here. Further language extensions to be introduced later on (see §3.3, chapter 5, chapter 6, chapter 7) may either provide concrete method and attribute definitions, or extend the primary command language itself. The former may never affect the integrity of the previous Isar grammar, since methods and attributes are always clearly delimited by their enclosing syntax. In contrast, derived commands could easily lead to syntactic conflicts, due to the non-modular nature of arbitrary grammar specifications.

In practice, we may achieve robust syntax extensions by restricting the "definitional" pattern for derived Isar commands essentially to a new keyword (with optional arguments) that expands to a sequence of existing command phrases; complete (local) proofs may be safely incorporated as well (e.g. see the syntax of **obtain** introduced in §5.3.1).

This disciplined way of building up the Isar language results in a syntactic environment that is both very clean and open-minded towards extensions.

3.3 Generic support for natural deduction

The abstract Isar framework covered so far still needs a few standard elements to enable users to express actual natural deduction concepts properly. This includes concrete context elements (like assumptions, see §3.3.1), attributes and proof methods (for composition and rule application, see §3.3.2), as well as basic derived commands that allow Isar texts to be written more fluently (see §3.3.3).

3.3.1 Context elements

Speaking in terms of the pure λ-calculus model of natural deduction, context elements closely correspond to abstraction. Recall that our primitive framework (§2.2) actually provides three different layers, with abstraction of the term language (function space $\Rightarrow$), universal parameters (quantification $\bigwedge$), and hypothetical proofs (implication $\Longrightarrow$).

No explicit contexts are required when building up the abstract syntax language, terms are built-up and type-checked directly (§2.2.1). Furthermore, universal parameters merely introduce local elements in the present proof (via **fix**, see §3.2.3), which are just generalized on export without imposing any additional hypotheses on the result (see also §5.2.1 for practical issues).

In contrast, actual logical hypotheses need to be taken care of specifically. Assumptions at the propositional level are "discharged" in a specific manner eventually, depending on the particular context element involved. To this end the basic operational model of Isar (§3.2.3) provides the generic **assm** primitive that is parameterized by an inference rule to determine the exact behavior. Based on this core element, we shall now define actual user-level context commands, namely **assume** for "strong" assumptions, **presume** for "weak" assumptions, **def** for local definitions, and **case** for invoking symbolic contexts. We first extend the primary Isar language (§3.2.1) as follows.

	assume (*name-atts*:)$^?$ *prop*$^+$ (**and** (*name-atts*:)$^?$ *prop*$^+$)*
$\mid$	**presume** (*name-atts*:)$^?$ *prop*$^+$ (**and** (*name-atts*:)$^?$ *prop*$^+$)*
$\mid$	**def** (*name-atts*:)$^?$ *var* $\equiv$ *term*
$\mid$	**case** *name-atts*

The conventions for default arguments (§3.2.1) are augmented for **def**: the standard *name* (of *name-atts*) shall be *x-def*, where x is the defined variable.

The three most basic Isar context elements are directly defined in terms of **assm**, via the rule schemes of *discharge#*, *discharge*, and *expand* given below.

assume q_1: $\vec{\varphi}_1$ **and** ... **and** q_n: $\vec{\varphi}_n$ =
  assm $\ll$*discharge#* $\vec{\varphi}_1$ ... $\vec{\varphi}_n\gg$ q_1: $\vec{\varphi}_1$ **and** ... **and** q_n: $\vec{\varphi}_n$

presume q_1: $\vec{\varphi}_1$ **and** ... **and** q_n: $\vec{\varphi}_n$ =
  assm $\ll$*discharge* $\vec{\varphi}_1$ ... $\vec{\varphi}_n\gg$ q_1: $\vec{\varphi}_1$ **and** ... **and** q_n: $\vec{\varphi}_n$

def q: $x \equiv t$ = **fix** x **assm** $\ll$*expand* $x \equiv t\gg$ q: $x \equiv t$

$$\frac{\Gamma \cup \{\vec{\varphi}\} \vdash \psi}{\Gamma \vdash \#\vec{\varphi} \Longrightarrow \psi}\ (\mathit{discharge\#}\ \vec{\varphi}) \qquad \frac{\Gamma \cup \{\vec{\varphi}\} \vdash \psi}{\Gamma \vdash \vec{\varphi} \Longrightarrow \psi}\ (\mathit{discharge}\ \vec{\varphi})$$

$$\frac{\Gamma \cup \{x \equiv t\} \vdash \psi}{\Gamma \vdash \psi[t/x]}\ (\mathit{expand}\ x \equiv t) \qquad \text{proviso: } x \text{ not free in } \Gamma \text{ or } t$$

These rules are clearly derivable within the basic logical framework (§2.2). First strengthen the local contexts to make sure they actually mention the additional assumptions as stated above. To get *discharge* iterate the $\Longrightarrow$*-intro* rule, the same works for *discharge#* (recall that "#" marks do not have any logical significance §2.4.1). In order to derive *expand* discharge the equality and generalize over x (which does not affect the context nor the right-hand side), then specialize with $[x/t]$ and apply modus ponens with reflexivity $\vdash t \equiv t$.

We also observe that any of these rules really get rid of the assumptions introduced beforehand: given "**assm** $\ll r \gg$ $\vec{\varphi}$" the rule r needs to map $\Gamma \cup \vec{\varphi} \vdash \psi$ to $\Gamma \vdash \psi'$, in order to guarantee that the Isar/VM interpretation process (§3.2.3) does not fail unexpectedly due to pending hypotheses.

The only difference of **assume** and **presume** is how the result is treated in a goal context (cf. §3.2.3). As indicated by the "#" markers, the new premises resulting from discharged strong assumptions are forced to unify with the original goal context, while the weak version simply leaves former hypotheses as new goals (cf. *refine* in §3.2.3). See also §5.2.1 for practical use of these Isar context elements.

The **case** command provides an abbreviation for several **fix**/**assume** elements. It invokes a local context according to the assignment of the current proof state σ. Given name a and $(\vec{x}, \lambda\vec{x}.\ \vec{\varphi}) = \mathit{get\text{-}cases}\ \sigma\ a$, we define **case** as follows.

$$\textbf{case}\ a\ [\vec{\alpha}] \quad = \quad \textbf{fix}\ \vec{x}\ \textbf{assume}\ a\ [\vec{\alpha}]\text{: } \vec{\varphi}$$

Recall that there is no separate Isar command to bind case names. Apart from the automatic *antecedent* case (§3.2.3, see also §5.2.5), further cases may be only introduced by suitable proof methods (like *cases* and *induct* covered in §5.4).

3.3.2 Methods and attributes

Method combinators

Proof methods may refer to arbitrary procedures that operate on primitive goal configurations. Potentially infinite sequences of results may represent multiple choices, e.g. from a fixed collection of rules, or enumeration of higher-order unifiers, or arbitrary internal proof search. Isar provides a standard collection of method combinators to compose such procedures in a simple fashion (analogous to regular expression operators).

m_1, m_2	sequential composition of methods
$m_1 \mid m_2$	alternative choice of methods
m?	try method
m+	repeat method (at least once)
succeed	identity method

Sequential composition "," and repeat "+" are most frequently used in practice; "*m?*" coincides with "$m \mid$ *succeed*", repeating a method including zero times may be expressed as "(*m+*)*?*", there is no separate "$m*$".

Note that excessive use of method combinators is actually an indication for highly operational expressions of unstructured proof scripts. In principle, a very long "script" of proof methods (m_1, ..., m_n) may be included in a single proof step. Structured Isar texts involve very simple method expressions only.

Basic methods

We introduce a few generic proof methods below. The syntactic category of *method* (§3.2.1) is extended as follows.

"(" *rule name-atts*$^+$ ")"
| *rule*
| *this*
| *assumption*
| –
| "(" *insert name-atts*$^+$ ")"
| "(" *unfold name-atts*$^+$ ")"
| "(" *fold name-atts*$^+$ ")"

The *rule* method provides a direct interface to the primary inference mechanism of the pure framework, namely higher-order resolution (cf. §2.4). The general form "(*rule* $\vec{a}$)" takes an explicit collection of rules to be tried from left to right. The (complete) list of chained facts *this* is taken into account as well. Given some rule r of $\vec{a}$, the method performs *goal* (r · *this*) on the current goal state (i.e. the rule is reduced by applying all facts in parallel, and the result applied to the first subgoal).

Omitting the argument of the *rule* method means to refer to "standard" rules declared in the present context. The attributes of *intro* and *elim* take care of appropriate rule declarations; *dest* declares eliminations presented in "projection" format, applying $\vdash A \Longrightarrow (A \Longrightarrow C) \Longrightarrow C$ first (e.g. $\vdash \forall x.\ P\ x \Longrightarrow P\ t$ is turned into $\vdash \forall x.\ P\ x \Longrightarrow (P\ t \Longrightarrow C) \Longrightarrow C$). Eliminations are tried before introductions, unless there are no chained facts at all, which is interpreted as a pure introduction pattern.

The *this* method applies all chained facts immediately (from left to right), without any rule in between; given chained facts *this* = [ϑ_1, ..., ϑ_n], the method performs (*goal* · ϑ_1) · ... · ϑ_n, i.e. just *goal* · *this* in the common case of a singleton chained fact.

The *assumption* method applies exactly one rule immediately, either a singleton chained fact, or one of the current *prems* of the proof context. New premises emerging from application of non-atomic rules are solved locally using *by-assumption* (§2.4.2).

The "−" method merely inserts all chained facts into the goal configuration (the do-nothing form with zero facts is mostly encountered in practice). The *insert* method inserts exactly the argument facts, but ignores the chained ones.

The *unfold* method normalizes a problem by a given collection of equalities (by repeated application of the substitution and extensionality rules of "≡", cf. §2.3). Its counterpart *fold* normalizes by swapped rules.

Incidently, the most common instances of the methods *this*, *assumption*, and "−" may be expressed in terms of the basic *rule* one as follows.

$$\begin{aligned} \mathit{this} &= (\mathit{rule} \vdash \bigwedge A.\ A \Longrightarrow A) \\ &\quad \text{for singleton facts} \\ \mathit{assumption} &= (\mathit{rule}\ \mathit{prems}) \\ &\quad \text{for empty facts and atomic assumptions} \\ - &= (\mathit{rule} \vdash \bigwedge \vec{A}\ C.\ \vec{A} \Longrightarrow (\vec{A} \Longrightarrow C) \Longrightarrow C) \\ &\quad \text{for atomic facts (same length as } \vec{A}) \end{aligned}$$

Basic operations on facts

The following attributes operate on theorems, without changing the current context. We extend the syntactic category *attribute* (§3.2.1) as follows

of term$^+$
| *OF name-atts*$^+$
| *THEN name-atts*
| *symmetric*

The attribute *of* provides the primitive operation of positional instantiation, as in the fact expression "a [*of* t_1 ... t_n]". The *OF* attribute performs "application" of a number of facts via higher-order resolution (cf. §2.4), as in the expression "r [*OF* a_1 ... a_n]"; the *THEN* attribute does the same, but exchanges the roles of operator and operand (which needs to be singleton), as in "a [*THEN* r]". The *symmetric* attribute swaps equality facts. The method expression "(*fold* $\vec{a}$)" is actually the same as "(*unfold* $\vec{a}$ [*symmetric*])".

3.3.3 Derived commands

We shall introduce a few very simple derived commands on top of the set of primitives provided so far. First of all, the primary syntax of Isar commands (cf. §3.2.1) is extended as follows.

lemmas (*name-atts* =)$^?$ *name-atts*$^+$
| **lemma** (*name-atts*:)$^?$ *prop*
| **hence** (*name-atts*:)$^?$ *prop*
| **thus** (*name-atts*:)$^?$ *prop*
| **from** *name-atts*$^+$ (**and** *name-atts*$^+$)*
| **with** *name-atts*$^+$ (**and** *name-atts*$^+$)*
| **by** *method method*$^?$
| **..**
| **.**

The default of the second method argument of **by** shall be *succeed* (just as for the **qed** constituent, cf. §3.2.1). The derived commands are defined as follows.

lemmas a $[\vec{\alpha}] = \vec{r}$	=	**theorems** a $[\vec{\alpha}$, *tag lemma*$] = \vec{r}$
lemma a $[\vec{\alpha}]$: φ	=	**theorem** a $[\vec{\alpha}$, *tag lemma*$]$: φ
hence	=	**then have**
thus	=	**then show**
from $\vec{q}$	=	**note** $\vec{q}$ **then**
with $\vec{q}$	=	**from** $\vec{q}$ **and** *this*
by m_1 m_2	=	**proof** m_1 **qed** m_2
..	=	**by** *rule*
.	=	**by** *this*

This collection of seemingly trivial shorthands has emerged from practical work performed in Isar, achieving significant improvements in both reading and writing of proof texts. Nevertheless, some care has to be taken whenever any further abbreviations shall be added, since excessive use of specific elements may eventually obscure the meaning of texts for casual readers.

Below we observe some further equalities of Isar commands due to the operational semantics (§3.2.3). First of all, we achieve alternative characterizations of basic operations involving the important **then** primitive.

from *this*	=	**then**
from *this* **have**	=	**hence**
from *this* **show**	=	**thus**

The following equality of **by** enables writers to take apart the individual phases of terminal proof steps in a fine-grained manner. This turns out as quite handsome in interactive development and debugging.

by m_1 m_2 = **apply** m_1 **apply** m_2 **apply** (*assumption*+)*?* **done**

3.4 Further concepts

In principle, the Isar/VM interpretation process presented so far (§3.2.3) is already sufficiently powerful to support high-level proof texts. The real implementation of Isabelle/Isar [Wenzel, 2001a] covers a few fine points, though,

that turn out as quite important to improve overall usability of Isar in practice. Below we briefly review these further issues, which are all outside of the core logical framework (§2.2).

3.4.1 Casual term abbreviations

Substantial parts of structured proofs consist of propositions (and sub-terms) given explicitly in the text. This is an important prerequisite to achieve human-readable presentations in the first place, unlike operational proof scripts that refer to internal goal transformations only. On the other hand, an excessive amount of concrete λ-terms in the text tends to degrade readability in its own right. Adequate syntactic abstractions turn out as a key issue of expressing formal reasoning succinctly. Isar already provides the concept of term abbreviations via the **let** command (§3.2.1). In practice, the extra overhead of separate abbreviation statements in the text is often too cumbersome.

Isabelle/Isar offers the additional **is** element that admits term abbreviations to be introduced on the fly. The basic syntax of Isar commands (§3.2.1) is augmented to include optional **is**-patterns after any occurrence of *term* or *prop*. The concrete syntax of these casual abbreviation forms is as follows:

term-patterns = "(" (**is** *term*)$^+$ ")"
prop-patterns = "(" (**concl**$^?$ **is** *prop*)$^+$ ")"

By using "(**is** p)" any term mentioned in the text may get immediately analyzed by (higher-order) matching against some pattern p. This essentially provides an immediate benefit in return of the duty to write explicit statements in the first place. For example, the annotated claim "**have** $a = b$ (**is** *?lhs* = *?rhs*)" enables succinct references to the (potentially unwieldy) terms a and b later on. Say the proof proceeds by an antisymmetry argument, then the body may just state "**show** *?lhs* $\leq$ *?rhs*" and "**show** *?rhs* $\leq$ *?lhs*". Such an abstract presentation may also clarify the actual proof pattern involved.

The **concl** specifier for *prop-patterns* indicates matching against the conclusion of a nested meta-level implication, e.g. "**have** $A \Longrightarrow C$ (**concl is** *?X*)" has the same effect as "**have** $A \Longrightarrow C$ (**is** - $\Longrightarrow$ *?X*)" (which uses the dummy pattern "-", cf. §3.2.3). The form "**have** φ (**concl is** *?thesis*)" documents the builtin binding of *?thesis* (cf. §3.2.3), but only if φ does not have any outer parameters. Likewise does "**have** φ (**concl is** - = ...)" represent the implicit argument binding of "...", at least in the common case of equational propositions. Recall that "..." technically acts like a schematic term variable (§3.2.3).

The full power of term abbreviations is exhibited by actual higher-order matching against complex statements. Here the main application is proof by induction (see also §5.4). The idiom of "**have** φ (**is** *?P n*)" essentially decomposes a statement $\varphi = \ldots\ n\ \ldots\ n\ \ldots$ into a predicate *?P* that abstracts over the occurrences of the fixed variable n in the original body. The standard procedure of enumerating higher-order unifiers in Isabelle [Paulson, 1989] ensures that *?P* really

abstracts over all occurrences of n (as is normally intended by the user). For example, this binding of *?P* enables succinct expressions of relevant statements of mathematical induction, with *?P* 0 for the base case, *?P* n for the induction hypothesis and *?P* (*Suc* n) for the conclusion of the step case. See also §5.4 for further advanced proof patterns.

Casual term abbreviations of Isabelle/Isar generally have the great virtue to reduce the need for special proof language constructs. For example, DECLARE [Syme, 1997a] [Syme, 1998] [Syme, 1999] requires separate provisions of "`ihyp` macros" for induction patterns. Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] includes dummy goal statements like `existence` or `uniqueness` to cover certain proof obligations arising in particular specification mechanisms (see also the related discussion in §7.5.1).

Moreover, analyzing term structure by higher-order matching serves as a viable replacement for "direct term manipulation", potentially with heavy user-interface support, as proposed in [Bertot and Thery, 1996] [Bertot *et al.*, 1997]. In contrast to interactive manipulations performed at run-time by the user, casual abbreviations in Isar may easily document advanced structural decompositions within the primary text, just by a few λ-term patterns.

3.4.2 Formal comments and antiquotations

From the perspective of recipients the ultimate intention of the Isar language is to describe formal documents, which consist of several theories with both specifications and proofs alike (cf. §3.2.1). In reality, such theory texts may also contain additional information outside the formal logic, like sectioning and informal explanations by the writer.

To this end, Isabelle/Isar [Wenzel, 2001a] provides several "markup commands" like **chapter**, **section**, **subsection**, and **text** (each taking a *text* argument). Concerning Isar itself, markup commands do not have any formal meaning, but are still part of the syntax of the language. Actual formal commands considered so far may also include "marginal comments" of the form "— *text*" that are related to particular entities, like individual declarations of **consts** (cf. §3.2.1).

consts
  $c_1 :: \tau_1$ — blah
  ...
  $c_n :: \tau_n$ — blub

Markup commands and marginal comments qualify as *formal comments* since there is an explicit relationship to formal elements, despite being devoid of any logical semantics themselves. Note that in commonly encountered "source comments" of existing languages there is usually no clear indication of the relationship with formal entities. Such comments may typically float around in the input text in a completely undisciplined manner.

The content of formal comments is ultimately passed to the document preparation front-end, which is (PDF)LaTeX in Isabelle/Isar. The text essentially consists of source code for the typesetting system, but is passed through a preparatory stage in Isabelle. Text specifications in Isabelle/Isar may contain embedded references to formal entities, such as well-typed terms and propositions, or theorems of the present context.

The concept of embedding interpreted parts into uninterpreted ("quoted") text is called *anti-quotation*, following existing concepts of LISP folklore. The basic syntax of anti-quotations in Isabelle/Isar is "@{*name args*}" (see also [Wenzel, 2001a]). In practice, the most important antiquotations are as follows.

@{*term* t}	well-typed term
@{*prop* φ}	well-typed proposition
@{*thm* $\vec{a}$}	facts (lists of theorems)
@{*text* s}	uninterpreted text

These anti-quotations process their argument within the formal *context* (§3.2.3), and emit the (checked) result into the document output in a pretty-printed form, just like any other Isar text. The degenerate *text* anti-quotation merely outputs the argument string directly, but treats mathematical symbols according to the Isabelle/Isar style (cf. §1.5) rather than raw LaTeX. Thus unchecked "formal" descriptions may use the same notation as real Isabelle/Isar objects, without demanding ad-hoc tweaks of the LaTeX math mode.

The overall effect of this seamless integration of formal and informal portions of text into a single Isabelle/Isar source considerably reduces the effort to report about theory developments in a consistent manner. Unlike existing approaches for "literate programming" (notably Knuth's WEB system) there is no need to filter the formal and informal views separately. In Isabelle/Isar, finished proof documents are output as a side-effect of formal proof processing, which in turn merely ignores certain parts of the text.

3.4.3 Type inference and polymorphism

In theory, we may easily pretend that all terms given in Isar proof texts are fully annotated with types, according to the type-checking rules of the underlying framework (§2.2). In practice, users may be spared from explicit type annotation chores via the well-established technique of automated type reconstruction, which is also known as *type inference*. This happens to be already present in the Isabelle/Pure implementation [Paulson and Nipkow, 1994].

It is important to note that such syntactic typing issues need not be considered within the actual logical framework, but only as an "accidental" feature of the implementation. This is analogous to the concept of *parsing*, which automatically reconstructs abstract syntax trees from user input presented in handsome concrete syntax. The theory of parsing was considered an issue of formal logic long ago, but it has lost its relevance eventually, as standard parser tools have

become well-established.

In order to make simple type inference available to Isar proof texts, we maintain the following additional fields of variable typings and used type variables in the proof *context* structure (§3.2.3).

typing-frees	::	*name* $\rightharpoonup$ *type*
typing-vars	::	*name* $\rightharpoonup$ *type*
used-types	::	*set of name*

Here the environments *typing-frees* and *typing-vars* determine the types for variables, as encountered in the proof text processed so far. Whenever new terms are prepared for inclusion in the text, we first perform standard (mutual) type inference within the present context of typings, and then use the resulting fully-typed term to extend these declarations. In order to guarantee most general results, type inference occasionally needs to invent "new" type variables; these are chosen as apart from the set of *used-types*, which is maintained accordingly.

The resulting discipline of type reconstruction proceeds sequentially from left to right through a given list of Isar commands (while observing block structure in the obvious manner). The scope of mutual type inference is limited to the arguments of each individual Isar command, e.g. "**assume** $\varphi_1 \ldots \varphi_n$" covers the propositions $\vec{\varphi}$ simultaneously. In practice, this scheme is fair enough most of the time, although rather annoying situations may occur whenever the inferred typing is more general than intended by the writer (due to lack of typing information from future text). This may cause unexpected failure of both further type checking and logical inferences (e.g. with rules that only work for specific type instances, probably due to overloading).

Certainly, writers may always fall back on explicit type annotations, without requiring readers of Isar proof texts to care very much. On the other hand, Isabelle users generally expect typing issues to be treated automatically behind the scenes. Any failures encountered here are apt to cause considerable confusion, until the actual problem is figured out by hand eventually.

The refined type inference scheme according to Hindley-Milner (also known as "let-polymorphism") [Hindley, 1969] [Milner, 1978] is slightly more flexible. This improved typing discipline needs to extend pure λ-calculus by a separate let-binder: in the term $\mathsf{let}\ x = t\ \mathsf{in}\ u$ the variable x is bound locally to t within the body u; the canonical conversion rule is $(\mathsf{let}\ x = t\ \mathsf{in}\ u) \longrightarrow u[t/x]$. This operational idea could be simulated in pure λ-calculus as a β-redex $(\lambda x.\ u)\ t$, but the key point of Hindley-Milner polymorphism is to have let as a separate primitive and provide a specific typing rule to achieve a localized form of schematic polymorphism.

Hindley-Milner polymorphism also extends the language of simple types by *type schemes*, which include "generalized" type variables that may be instantiated arbitrarily. In the literature this is usually represented by a flat prefix of universal type quantifiers "$\forall\alpha$". In Isabelle/Pure we may express the same idea

via schematic type variables $?\alpha$ (§2.2.1), so the canonical typing rule for let expressions becomes this:

$$\frac{\begin{array}{cc} & [x\colon \hat{\sigma}] \\ & \vdots \\ t\colon \sigma & u\colon \tau \end{array}}{(\mathsf{let}\ x = t\ \mathsf{in}\ u)\colon \tau}$$

Here the inferred type σ of the local binding is replaced by a most general type scheme $\hat{\sigma}$ when type-checking the body; all fixed type variables α in σ that do not occur in any fixed term variable of the context become schematic ones $?\alpha$ in $\hat{\sigma}$. Thus the typing of x may be instantiated in the body later on.

In order to incorporate this refined typing discipline into the Isar/VM interpretation process, we merely introduce another field in the proof *context*.

fixed-types :: *set of name*

During Isar proof processing, the *fixed-types* component is maintained to hold the set of type variables that are still considered "fixed", due to occurrences in types of term variables that are manifest in the previous text (bound variables and constants do not matter here). Now we only need to identify suitable kinds of let-bindings in Isar where type schemes may be generalized locally.

Recall that the Isar/VM interpretation process (§3.2.3) may be understood as a certain policy for composing proofs according to general λ-calculus concepts (cf. §2.2, and see §5.2 for the user-level view). From this perspective, it is easy to see that two kinds of Isar context elements qualify as polymorphic let binders: term abbreviations introduced via **let** (§3.2.1) or **is** (§3.4.1), and local facts emerging from the primitives **note**, **have**, or **show** (§3.2.1).

Note that polymorphic treatment of proper abstraction elements like **fix** and **assume** (§3.3.1), would demand actual "polymorphic λ-calculus", such as $\lambda 2$ or beyond (e.g. see [Barendregt, 1992]), which would quickly lead into a substantially more complex situation (with undecidable problems). On the other hand, variables introduced by unconstrained **fix** statements in isolation need not be typed until their actual occurrence in the subsequent text; the stages of binding and typing of variables may be kept separate without further ado.

Local definitions "**def** $x \equiv t$" (§3.3.1) could in principle be treated as polymorphic, too, but our present formulation within Isar makes **def** appear like the monomorphic "**fix** x **assume** $x \equiv t$". This minor drawback is hard to fix in reality, mainly because the Isabelle/Pure implementation [Paulson and Nipkow, 1994] does not admit fixed variables at different type instances within theorems. Note that **let** and **is** are significantly more important in practice (§3.4.1).

In order to get an idea of how Hindley-Milner typing works out in Isar, we consider the following synthetic example.

let *?f* = $\lambda x.\ x$
  — *?f* :: $?\alpha \Rightarrow ?\alpha$
have *?f ?f* = $(\lambda x.\ x)$
  — first occurrence *?f* :: $(\alpha \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \alpha)$
  — second occurrence *?f* :: $\alpha \Rightarrow \alpha$
  by (*rule refl*)
  — *this* = $\vdash (\lambda x :: ?\alpha.\ x) = (\lambda x :: ?\alpha.\ x)$

Here *?f* is bound to the identity function, with types being generalized fully. When checking the subsequent goal statement the typing of *?f* is instantiated twice, and held fixed during the proof. In the resulting theorem types are again fully generalized.

In reality, actual polymorphic proof texts are rarely encountered at all. The key virtue of Hindley-Milner polymorphism in Isar is to achieve a well-defined discipline that is able to amend (most of) the problems with overly general inferred types due to incremental processing of the text. Recall that the original problem has been caused by lack of syntactic relationship of previous proof text with potential follow-up material. In such a situation naive type inference would invent new (fixed) type variables, expressing unrestricted generality. The refined typing discipline due to Hindley-Milner is able to generalize these variables for the very same reason they got introduced in the first place, which gives subsequent typing stages a chance to instantiate these as required.

Experience with Isabelle/Isar shows that this fine-tuned discipline is really able to relieve proof writers from most typing issues in practice. Certainly, the situation would be much simpler with batch-mode proof processing, where the whole text may be covered at once, such as in DECLARE [Syme, 1997a] [Syme, 1998] [Syme, 1999]. Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] even requires users to give type annotations themselves.

Chapter 4

Example: First-Order Logic

We present a formulation of intuitionistic first-order logic as canonical instantiation of the generic Isar framework. This demonstrates how the existing tradition of object-logic declarations in the Isabelle environment may be extended to cover readable presentations of formal proofs as well. Handling both specifications and proofs in a high-level manner, Isabelle/Isar qualifies as a truly complete logical framework.

By the example of first-order logic we also discuss the most basic techniques of basic natural deduction proofs, both within Isar as well as some other systems.

4.1 Formal development

theory *First-Order-Logic = Pure*:

The present theory development introduces single-sorted intuitionistic first-order logic with equality. We are giving common abstract and concrete syntax, basic axioms, definitions and derived rules, together with readable formal proofs of standard derived rules and further examples.

4.1.1 Syntax

There are two categories of higher-order abstract syntax for the object language under consideration: i for "individuals" and o for "object statements"; the latter shall be implicitly used as a meta-logical judgment of derivable sentences.

typedecl i
typedecl o

judgment
Trueprop :: $o \Rightarrow prop$ (- 5)

4.1.2 Propositional logic

The basic propositional connectives are axiomatized canonically as follows.

consts
false :: o ($\bot$)
imp :: $o \Rightarrow o \Rightarrow o$ (**infixr** $\longrightarrow$ 25)
conj :: $o \Rightarrow o \Rightarrow o$ (**infixr** $\wedge$ 35)
disj :: $o \Rightarrow o \Rightarrow o$ (**infixr** $\vee$ 30)

axioms
falseE [*elim*]: $\bot \Longrightarrow A$

impI [*intro*]: $(A \Longrightarrow B) \Longrightarrow A \longrightarrow B$
mp [*dest*]: $A \longrightarrow B \Longrightarrow A \Longrightarrow B$

conjI [*intro*]: $A \Longrightarrow B \Longrightarrow A \wedge B$
$conjD_1$: $A \wedge B \Longrightarrow A$
$conjD_2$: $A \wedge B \Longrightarrow B$

disjE [*elim*]: $A \vee B \Longrightarrow (A \Longrightarrow C) \Longrightarrow (B \Longrightarrow C) \Longrightarrow C$
$disjI_1$ [*intro*]: $A \Longrightarrow A \vee B$
$disjI_2$ [*intro*]: $B \Longrightarrow A \vee B$

The following derived rule of simultaneous conjunction elimination is usually more convenient to use than referring to the individual projections separately.

theorem *conjE* [*elim*]: $A \wedge B \Longrightarrow (A \Longrightarrow B \Longrightarrow C) \Longrightarrow C$
proof −
assume *ab*: $A \wedge B$
assume *r*: $A \Longrightarrow B \Longrightarrow C$
show C
proof (*rule r*)
from *ab* **show** A **by** (*rule* $conjD_1$)
from *ab* **show** B **by** (*rule* $conjD_2$)
qed
qed

Furthermore, we introduce the derived concepts of plain truth and negation.

constdefs
true :: o ($\top$)
$\top \equiv \bot \longrightarrow \bot$
not :: $o \Rightarrow o$ ($\neg$ - [40] 40)
$\neg A \equiv A \longrightarrow \bot$

theorem *trueI* [*intro*]: $\top$
proof (*unfold true-def*)
  show $\bot \longrightarrow \bot$ **..**
qed

theorem *notI* [*intro*]: $(A \Longrightarrow \bot) \Longrightarrow \neg\, A$
proof (*unfold not-def*)
  assume $A \Longrightarrow \bot$
  thus $A \longrightarrow \bot$ **..**
qed

theorem *notE* [*elim*]: $\neg\, A \Longrightarrow A \Longrightarrow B$
proof (*unfold not-def*)
  assume $A \longrightarrow \bot$ **and** A
  hence $\bot$ **.. thus** B **..**
qed

4.1.3 Equality

Equality of individuals is axiomatized in a high-level manner, using reflexivity and substitution as primitive. The remaining equivalence properties are easily established as derived rules. Congruence properties are already covered by the substitution rule, so these are not stated explicitly.

consts
  equal :: $i \Rightarrow i \Rightarrow o$ (**infixl** = 50)

axioms
  refl [*intro*]: $x = x$
  subst: $x = y \Longrightarrow P(x) \Longrightarrow P(y)$

theorem *trans*: $x = y \Longrightarrow y = z \Longrightarrow x = z$
  by (*rule subst*)

theorem *sym* [*elim*]: $x = y \Longrightarrow y = x$
proof −
  assume $x = y$
  from *this* **and** *refl* **show** $y = x$ **by** (*rule subst*)
qed

4.1.4 Quantifiers

Within the underlying logical framework quantifiers are simply certain operators on predicates, while concrete syntax for "binders" recovers commonly used notation.

consts
  All :: $(i \Rightarrow o) \Rightarrow o$ (**binder** $\forall$ 10)

$Ex :: (i \Rightarrow o) \Rightarrow o$ (**binder** $\exists$ 10)

axioms
$allI$ [*intro*]: $(\bigwedge x.\ P(x)) \Longrightarrow \forall x.\ P(x)$
$allD$ [*dest*]: $\forall x.\ P(x) \Longrightarrow P(a)$

exI [*intro*]: $P(a) \Longrightarrow \exists x.\ P(x)$
exE [*elim*]: $\exists x.\ P(x) \Longrightarrow (\bigwedge x.\ P(x) \Longrightarrow C) \Longrightarrow C$

Here is a simple example of reasoning with quantifiers; the statement has been taken from a [Paulson, 2001a].

lemma $(\exists x.\ P(f(x))) \longrightarrow (\exists y.\ P(y))$
proof
  assume $\exists x.\ P(f(x))$
  thus $\exists y.\ P(y)$
  proof
    fix x **assume** $P(f(x))$
    thus *?thesis* **..**
  qed
qed

Subsequently, we establish another well-known result of quantifier reasoning (naturally the converse statement does *not* hold in general).

lemma $(\exists x.\ \forall y.\ R(x,\ y)) \longrightarrow (\forall y.\ \exists x.\ R(x,\ y))$
proof
  assume $\exists x.\ \forall y.\ R(x,\ y)$
  thus $\forall y.\ \exists x.\ R(x,\ y)$
  proof
    fix x **assume** a: $\forall y.\ R(x,\ y)$
    show *?thesis*
    proof
      fix y **from** a **have** $R(x,\ y)$ **..**
      thus $\exists x.\ R(x,\ y)$ **..**
    qed
  qed
qed

end

4.2 Discussion

4.2.1 Generic proof support for object-logics

Our basic formulation of FOL as an Isabelle object-logic closely follows a similar example given in [Paulson, 2001a], while proofs have been expressed in the Isar

proof language, rather than traditional tactic scripts.

Purely declarative specification of logical syntax and axioms, together with derived rules expressed as explicit theorem statements within the meta-logic, have been the key concepts of the basic Isabelle/Pure framework from its very beginnings [Paulson, 1986] [Paulson, 1989] [Paulson, 1990]. In contrast, the original tradition of the "LCF" and "HOL" family of systems would have required explicit programming of derived rules as functional programs written in the ML "meta-language"; see also the historical account given in [Gordon, 2000].

From this perspective, the Isar approach to readable proof documents continues this mission to overcome low-level technical presentations of formal logic. The framework of Isabelle/Pure + Isar is able to support all of logical syntax, axioms, derived rules, *and* readable formal proof texts in a declarative manner.

As an illustration of the different conceptual levels of proof construction in traditional Isabelle/Pure versus Isabelle/Isar, reconsider the very same FOL example given in [Paulson, 2001a]. Before presenting any proof scripts, Paulson sets out on a lengthy exposition of a number of internal features of the Isabelle system, covering details about higher-order unification, composition of rules via higher-order resolution ("back-chaining"), lifting of rules into logical contexts, representation of proof states as rules, and basic concepts of the tactic language.

While these technical issues are still present in the Isar framework, they are mostly covered "under the hood" — only the higher-level concepts of the Isar proof language are exposed to recipients. Consequently, we have been able to present our theory in a rather casual manner with both specifications and proof texts. Recipients familiar with natural deduction techniques should be basically able to *read* these proofs without much further explanations required. On the other hand, additional insights into the formal proof language would certainly be needed to *write* proof texts of this kind. Recall that Isar follows the principle of "primacy of readability over writability" (§1.3).

Another notable issue is that of automated proof tools, especially as it is completely absent from the present example! While the Isabelle environment [Paulson and Nipkow, 1994] provides a number of powerful proof tools, such as the Simplifier and Classical Reasoner, these have not been used here (new object-logics would have to configure these tools explicitly in the first place).

Apart from some explicit proof method specifications of unfolding definitions and applying basic rules, we have merely used a simple (default) proof tool which supports single natural-deduction rule applications in an implicit manner (§3.3.2): rules have been determined according to the theory declaration, which includes a few hints such as "[*intro*]", "[*elim*]", "[*dest*]". Less pure applications (e.g. chapter 9 and chapter 10) certainly demand additional advanced tools (see also §7.3).

As a general principle, Isar has been made independent of any particular notion of automated reasoning (§1.3), while being able of any such tools that happen to be available. This is in notable contrast to common believe on high-level proof

checking. Consequently, Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] has certain notions of "obvious" inference steps hardwired into the proof checker. Likewise, much of the discussion of the "Mizar mode for HOL" [Harrison, 1996b] is dedicated to the issue of justification of proof steps by the Meson proof procedure; substantial parts of the work on the "structured proof language" SPL [Zammit, 1999a] [Zammit, 1999b] covers first-order automated reasoning.

In contrast, Isabelle/Isar demonstrates both that meaningful applications may be conducted with plain higher-order resolution (in single steps) alone, and that arbitrarily complex tools may be incorporated in a non-intrusive manner (see also §7.3).

4.2.2 Natural deduction schemes

The present first-order theory provides a number of rules for the canonical treatment of logical connectives according to natural deduction (cf. [Gentzen, 1935]). Together with the appropriate declarations of their role as introductions and eliminations (or destructions), this basic setup already enables us to write Isar proof texts that directly correspond to these natural deduction schemes.

The resulting presentation illustrates the most basic techniques of writing Isar proof texts. At the same time it also provides a nice textual explanation of how natural deduction reasoning works in the first place.

The trivial introduction of $\top$ and elimination of $\bot$:

have $\top$ **..**

assume $\bot$
hence A **..**

Introduction of $\neg$, and its elimination ("proof by contradiction"):

have $\neg\ A$
proof
  assume A
  thus $\bot$ ⟨*proof*⟩
qed

assume $\neg\ A$ **and** A
hence B **..**

Canonical $\longrightarrow$ introduction, and elimination ("modus ponens"):

have $A \longrightarrow B$
proof
  assume A
  show B ⟨*proof*⟩

qed

assume $A \longrightarrow B$ **and** A
hence B **..**

Introduction of $\wedge$, its two projections, as well as simultaneous elimination:

have $A \wedge B$
proof
  show A $\langle proof \rangle$
  show B $\langle proof \rangle$
qed

assume $A \wedge B$
hence A **..**

assume $A \wedge B$
hence B **..**

assume $A \wedge B$
hence C
proof
  assume A **and** B
  thus C $\langle proof \rangle$
qed

Elimination of $\vee$ (i.e. propositional case split), as well as its two introductions:

assume $A \vee B$
hence C
proof
  assume A
  thus C $\langle proof \rangle$
next
  assume B
  thus C $\langle proof \rangle$
qed

assume A
hence $A \vee B$ **..**

assume B
hence $A \vee B$ **..**

The basic equality rules of reflexivity (introduction), substitution (elimination), and the derived forms of transitivity and symmetry:

have $x = x$ **..**

assume $x = y$ **and** $P(x)$
hence $P(y)$ **by** (*rule subst*)

assume $x = y$ **and** $y = z$
hence $x = z$ **by** (*rule trans*)

assume $x = y$
hence $y = x$ **..**

Canonical introductions and eliminations of the $\forall$ and $\exists$ quantifiers:

have $\forall x.\ P(x)$
proof
  fix x
  show $P(x)$ $\langle proof \rangle$
qed

assume $\forall x.\ P(x)$
hence $P(a)$ **..**

assume $P(a)$
hence $\exists x.\ P(x)$ **..**

assume $\exists x.\ P(x)$
hence C
proof
  fix x **assume** $P(x)$
  thus C $\langle proof \rangle$
qed

While the above proof schemes follow common expositions of natural deduction rules quite closely (e.g. [Thompson, 1991]), in actual applications they are not always as "natural" as advertised. In particular, the equality rules and $\exists$ elimination are typical candidates for further refinements.

An important point of the Isar language concept is that the course of reasoning may be rearranged in numerous ways, as well will see in further examples later on. Furthermore, Isar supports a number of derived concepts that address further inconveniences of pure natural deduction encountered in realistic proofs. These advanced techniques include generalized elimination schemes (see chapter 5), and proper support for equational reasoning via calculations (chapter 6).

4.2.3 Declarative versus operational theorem proving

We shall now investigate a few basic issues of "declarative" proof texts versus "operational" proof scripts, as far as plain natural deduction is concerned.

The fundamental aspects of proof construction in a natural deduction framework like Isabelle/Pure are that of *statements* (propositions), *rules* (probably with instantiations), and *composition* of partial results (determining the overall proof structure). Roughly speaking, declarative proofs prefer to state propositions explicitly and provide rich text structure, rather than specify rules of inference; on the other hand, operational scripts merely give rules (or other proof method specifications). From this perspective, declarative versus operational proofs would be exactly dual to each other, by emphasizing complementary aspects of formal proofs.

Nevertheless, this characterization turns out to be slightly oversimplifying. In practice, Isar proofs may be declarative or operational to rather different degrees. Actual readability of the result depends on many factors, including the intention of the writer addressing a certain audience of readers. This may demand to highlight either "declarative" or "operational" aspects of the reasoning, depending on the present context. Just consider the example of large induction proofs (§5.4) involving inductively defined sets (§7.2.1). Here it is usually preferable to suppress explicit propositions from the text, but give a quasi-operational specification of the induction scheme (via a proof method) plus some structure on the emerging cases (see chapter 10 for typical examples). Thus we may gain readability by shifting the focus from explicit propositions over to proof methods and very abstract structure.

For the moment, we stay within plain natural deduction and illustrate the most basic declarative and operational techniques of Isar proof construction. Several variations for the propositional fact $A \wedge B \longrightarrow B \wedge A$ will be discussed.

Proof texts

Our first version follows more or less the standard idiom of plain natural deduction in Isabelle/Isar, with mixed forward and backward reasoning, cf. the basic introduction and elimination schemes given in §4.2.2.

lemma $A \wedge B \longrightarrow B \wedge A$
proof
  assume $A \wedge B$
  thus $B \wedge A$
  proof
    assume B **and** A
    thus *?thesis* **..**
  qed
qed

Apparently, we have been able to complete the proof without ever naming rules explicitly, or even just local facts. The deeper reason for this is twofold. First, we have explicitly stated assumptions and intermediate claims by giving an actual proposition as a term. This may sound like a rather obvious thing to do, but in

the tactical theorem proving tradition one would attempt to suppress explicit terms as much as possible. Second, our proof has been quite detailed about its overall structure, although this information is given quite implicitly, by nesting of sub-proofs, and performing suitable "gestures" to indicate what to do next. In particular, we have indicated forward chaining from existing facts as opposed to mere backward reasoning where appropriate (via **then** as involved in **thus**).

In order to see better how this kind of implicit processing of basic inferences works out in detail, we shall now expand the above proof further, until sufficient operational detail is exhibited. First of all, a few basic abbreviations have been used routinely; by expanding these we arrive at a slightly more explicit scheme.

lemma $A \wedge B \longrightarrow B \wedge A$
proof (*rule*)
  assume $A \wedge B$
  then show $B \wedge A$
  proof (*rule*)
    assume B **and** A
    then show *?thesis* **by** (*rule*)
  qed
qed

By default, the *rule* method figures out the actual rule to be used implicitly (cf. §3.3.2), which is usually quite easy based on the explicit goal statement given, together with the indication for forward chaining of facts (using **then**). The rules determined here are named explicitly in the next version.

lemma $A \wedge B \longrightarrow B \wedge A$
proof (*rule impI*) — canonical introduction of $\longrightarrow$
  assume $A \wedge B$
  then show $B \wedge A$ — canonical elimination of $\wedge$
  proof (*rule conjE*)
    assume B **and** A
    then show *?thesis* **by** (*rule conjI*) — canonical introduction of $\wedge$
  qed
qed

We see that canonical introductions may be simply performed by stating a goal an performing a single default proof step; likewise, canonical elimination works by indicating a fact for forward chaining, as before. As we may see in the third step above, forward chaining may result in introduction steps as well, if the proof is forced to be finished afterwards; here introductions are tried after all eliminations (cf. §3.3.2), so this scheme would still work if facts B and A provided separate logical structure, which may have become eliminated as well.

The subsequent version is even more obfuscated, as we include explicit instantiations of rules as well. Certainly, we would normally leave it to the builtin unification of Isabelle [Paulson and Nipkow, 1994] to work out such syntactic details.

lemma $A \wedge B \longrightarrow B \wedge A$
proof (*rule impI* [*of* $A \wedge B$ $B \wedge A$])
  assume $A \wedge B$
  then show $B \wedge A$
  proof (*rule conjE* [*of* A B $B \wedge A$])
    assume B **and** A
    then show *?thesis* **by** (*rule conjI* [*of* B A])
  qed
qed

From the highly redundant proof texts above we also see that that Isabelle/Isar proof checking actually involves a twofold book-keeping process, with explicit statements and structure on the one side, and operational steps on the other side. While, the common Isar idiom usually prefers the former (declarative) parts over the latter (operational) ones, we may as well choose otherwise — the Isar framework is sufficiently flexible to support rather "improper" uses of the language. This liberal attitude certainly demands some taste and distinction of the user, lest the system be abused in uncouth manners.

Proof scripts

The next version follows a purely operational style of tactical proving, by expressing the main reasoning steps via a string of proof methods alone; only the main statement is left as an explicit proposition.

lemma $A \wedge B \longrightarrow B \wedge A$
  by (*rule impI*, *erule conjE*, *rule conjI*)

Apparently, this way of emulating traditional tactic scripts stretches the Isar method language (cf. §3.3.2) a bit far, using sequential composition of methods to express the whole course of reasoning by a single command. Even worse, that form would be rather impractical for interactive development and debugging, since the **by** command succeeds (or rather fails) in a single atomic transition of the Isar/VM interpreter (cf. §3.2.3 and §3.3.3).

Proof scripts are more appropriately represented via "improper" proof commands **apply** and **done** (cf. §3.2.1), which support step-by-step goal refinements and do not refer to any implicit Isar reasoning steps (such as the implicit finishing by assumption, cf. §3.2.3).

lemma $A \wedge B \longrightarrow B \wedge A$
  apply (*rule impI*)
  apply (*erule conjE*)
  apply (*rule conjI*)
   apply *assumption*
  apply *assumption*
  done

While the last three commands above could be expressed as an immediate proof "**.**" as well, unstructured scripts better use the null proof terminator **done** and name any required assumption steps explicitly — this improves robustness and maintainability.

We see that operational scripts as above mostly consist of method specifications; explicit propositions only occur in the very beginning, structural hints are very limited (e.g. *erule* used instead plain *rule* basically amounts to a limited form of the forward chaining gesture as indicated by **then** in proper Isar proof texts).

Furthermore, our proof scripts do not provide any immediate information about the inherent tree structure of proof problems emerging by new subgoals as we proceed. While it is customary to indent script commands accordingly, this merely counts as a "comment" that is not processed formally. Note that this particular problem is specific to Isabelle [Paulson and Nipkow, 1994], being the cost of a very flexible approach to internal goal addressing. Proof scripts in some other systems reflect the subgoal structure directly in the text, e.g. Coq [Barras *et al.*, 1999] and HOL [Gordon and Melham, 1993] provide separate combinators to fork a script into several sub-scripts, in order to address the corresponding goals separately.

In proper Isar proof texts on the other hand, we may benefit from Isabelle's flexible scope on internal goals, without suffering from its potential problems — the structure of Isar sub-proofs is already determined by explicit local statements in the text (**have**, **show** etc.). A common pattern is to establish such local claims directly by an atomic proof of "**by** m_1 m_2", involving a tiny script of two methods only. Here the initial method m_1 (used with any chained facts) splits the original goal into a number of subgoals, and the terminal method m_2 solves any number of these, probably leaving a few trivial ones to be finished implicitly by assumption at the very end of this local proof.

Another notable issue is that of bringing explicit propositions back into proof scripts. Existing systems such as Isabelle [Paulson and Nipkow, 1994] and HOL [Gordon and Melham, 1993] do provide a number of tactics that take term specifications as additional arguments, e.g. `subgoal_tac` to simulate a local claim within the present goal state (resembling Isar's **have** to some extent). Nevertheless, such tactical elements are only rarely used in tactical proving (cf. the discussion in [Simons, 1996]). Experts of tactical proving occasionally even include comments with excuses about mentioning intermediate propositions during the course of reasoning!

As has been shown by longterm experience with tactical proof scripts, there are indeed some good reason for avoiding explicit quoting of terms: otherwise scripts may become "unstable" and hard to maintain afterwards. Seen from the Isar perspective, the problem is that of undisciplined intermixing of static and dynamic parts of proof states (cf. *context* versus *goal* in §3.2.3). Explicit propositions in scripts belong to the static text, but somehow need to refer to the dynamic goal state emerging from several tactics applied so far. Due to the very nature of common tactics, that dynamic result is very hard to predict, and

easily mutates under minor changes of theory definitions and declarations. Thus parts of a slightly obscure dynamic state would intrude the static text, which may be both quite surprising to the reader and easily break down existing proof scripts later on.

Unfortunately, in realistic applications even the most tuned operational proof scripts do have to mention explicit terms occasionally, such as in explicit instantiation of non-trivial rules like $\forall$ elimination, $\exists$ introduction, or induction schemes (cf. `res_inst_tac` in classic Isabelle [Paulson, 2001b] and *rule-tac* in the script emulation of Isabelle/Isar [Wenzel, 2001a]). Thus dynamically generated local parameters with accidental names such as *x xa xb* easily intrude proof scripts in an uncouth manner. In principle, effects like this would better have been accommodated by more careful usage of tactics (e.g. including *rename-tac* to fix parameter names). On the other hand, this kind of odd behavior of proof scripts is generally accepted as a matter of fact in tactical theorem proving.

Apart from using propositions as part of the control script, one may as well consider to restrict them to documentation purposes of the dynamic evolution of the internal goal state, in order to gain some accessibility of the result to casual readers. [Cohn, 1995] proposes this kind of support of "proof accounts" for HOL [Gordon and Melham, 1993]. That system includes a separate copy of the basic collection of HOL tactics to produce suitable output of current changes of the goal state. Any approach like this is faced with the problem of reducing the proof state information to relevant bits. The raw goal state at arbitrary intermediate positions of typical proof scripts easily becomes quite large (up to several printed pages in extreme cases), but only a few local differences to the previous steps represent the actual progress made.

Subsequently, we give a trace of the dynamic goal states encountered during our present example proof script. Even in this rather trivial case the raw output is already cluttered by much irrelevant information.

lemma $A \wedge B \longrightarrow B \wedge A$
— subgoals: 1. $A \wedge B \longrightarrow B \wedge A$
apply (*rule impI*)
— subgoals: 1. $A \wedge B \Longrightarrow B \wedge A$
apply (*erule conjE*)
— subgoals: 1. $A \Longrightarrow B \Longrightarrow B \wedge A$
apply (*rule conjI*)
— subgoals: 1. $A \Longrightarrow B \Longrightarrow B$ 2. $A \Longrightarrow B \Longrightarrow A$
apply *assumption*
— subgoals: 1. $A \Longrightarrow B \Longrightarrow A$
apply *assumption*
— subgoals: *No subgoals*!
done

One of the key observations of readable Isar proof descriptions is that the general course of reasoning is more adequately represented as a static text, without ever referring to dynamic goal information directly (proper Isar elements do not allow

this in the first place). Thus we are enabled to replace an internal dynamic trace of goals (i.e. a long list of large states) by a single static text of reasonable size. By this general approach we may expect readable proof representations that scale up to large applications as well.

Another lesson learned here is that the Isar framework is very liberal, allowing many ways to conduct formal reasoning. For those who know how to use the system properly, this freedom provides powerful means for interactive development and experimentation, as well as unusual presentations of final results. Nevertheless, Isabelle/Isar texts may be written in almost arbitrarily bad style. The Isar design deliberately suffers some degree of potential abuse as a price to be paid for freedom; recall the principle of "*abusus non tollit usus*" (§1.3).

4.2.4 Further expressions of natural deduction

We now change the perspective from Isabelle/Isar to a few other systems and notations for plain natural deduction. By comparing different expressions of natural deduction with corresponding Isar proof texts we gain further understanding of both those alternative systems as well as Isar itself.

After the original formulation of [Gentzen, 1935], plenty of alternative systems and notations have been devised to represent natural deduction proofs adequately. Such efforts include various forms of diagrams with lines or boxes to lay out trees of inferences (cf. the basic formats encountered in [Jape], for example), or even more advanced graphs and pictures rendered as proposed in [Barwise and Etchemendy, 1995] [Barwise and Etchemendy, 1998].

Several more recent systems (ProveEasy [Burstall, 1998], Mizar-Light [Wiedijk, 2000], and Tutch [Abel *et al.*, 2001]) have rediscovered the value of plain textual representations as a primary format for proofs. We argue that complex graphical representations are limited to small examples of formal logic or very special applications only. Diagrammatic presentations are inherently restricted in size and structural complexity by their physical appearance. It is quite hard to oversee large pictures, and even unclear where to start "reading" of a non-linear representation in the first place. Our claim is backed by the observation that non-textual proof formats are rarely encountered in large applications of formal logic. Also note that traditional mathematics works with linear texts most of the time as well (with the notable exception of highly abstract diagrammatic proofs encountered in category theory, for example).

Plain lambda-calculus

As far as *primitive* proof objects are concerned, natural deduction is certainly most adequately represented by typed λ-terms (cf. §2.2). Recent work on the Isabelle inference kernel [Berghofer and Nipkow, 2000] even provides a concrete programming interface based on this representation, supporting tools that

need to externalize primitive proofs from the core system (e.g. external checkers, storage facilities for primitive theories and theorems, facilities for proof-carrying code). Nevertheless, the internal representation of primitive proofs is independent of the issue of readable *primary* proof formats in Isar (cf. §1.4).

In Coq [Barras *et al.*, 1999] the notion of internal proof term has been tied to λ-calculus from the very beginning. The user experience of interactive development of goal-oriented proof scripts does not directly expose these foundations under normal circumstances, as is illustrated by the following example.

```
Goal (A, B: Prop)(A /\ B) -> (B /\ A).
  Intros a b ab.
  Induction ab.
  Split.
  Assumption.
  Assumption.
Save example1.
```

Nevertheless, Coq admits users to construct proofs directly by giving λ-terms as well. These may be either provided as definitions of proof terms, or immediately included in proof scripts; the latter feature is typically used by expert users to perform small forward inferences in a casual manner.

```
Definition example2 := [A, B: Prop; ab: A /\ B]
  (and_ind A B B /\ A [a: A; b: B](conj B A b a) ab).

Goal (A, B: Prop) A /\ B -> B /\ A.
  Exact [A, B: Prop; ab: A /\ B]
    (and_ind A B B /\ A [a: A; b: B](conj B A b a) ab).
Save example3.
```

Ad-hoc reasoning like this may be simulated in Isabelle/Isar only to a limited extent, using theorem expressions with basic attributes like *OF* (composition) or *of* (instantiation) (cf. §3.3.2). While composition covers plain application as well there is no standard attribute for abstraction; below we use proper Isar proof context commands instead.

lemma $A \wedge B \longrightarrow B \wedge A$
proof
  assume *ab*: $A \wedge B$
  show $B \wedge A$
  proof (*rule conjE* [*OF ab*])
    assume *a*: A
    assume *b*: B
    show *?thesis*
      by (*rule conjI* [*OF b a*])
  qed
qed

On the other hand, the very aim of the Isar proof language is to replace the primitive notion of λ-terms by a primary proof format that is more accessible to readers. Theorem expressions with attributes as encountered above are only rarely used in proper Isar proof texts at all.

The Agda system [Agda] [Coquand and Coquand, 1999] is positioned as a reformed version of Coq, being based on a different version of typed λ-calculus internally, but with a similar system philosophy. The default user experience of Agda is quite different from Coq, exposing its λ-calculus foundations directly to the primary proof presentation format, which resembles an explicitly typed higher-order functional programming language. Consider the following Agda version of our running example of $A \wedge B \longrightarrow B \wedge A$.

```
example (A::Prop)(B::Prop) :: Implies (And A B) (And B A)
  = ImpliesIntro (And A B) (And B A)
      (\(ab::And A B) ->
       AndIntro B A (AndElim2 A B B ab (\(b::B) -> b))
         (AndElim1 A B A ab (\(a::A) -> a)))
```

Alfa is a separate graphical proof editor for Agda, which has been recently enhanced to support natural language input and output [Hallgren and Ranta, 2000] as well. Using the Alfa user-interface, λ-terms may be drawn in two-dimensional diagrams according a well-established format of natural deduction proof trees (cf. the textbook exposition of [Thompson, 1991]). A typical proof presentation of Alfa looks as follows (it is important to note that the formal structure of the underlying Agda proof is quite different from the previous one).

$$\dfrac{\lambda ab \rightarrow \dfrac{\dfrac{\dfrac{\overline{A \wedge B}\ ab}{B}\wedge E_2 \quad \lambda b \rightarrow \dfrac{}{B}\ b \qquad \dfrac{\dfrac{\overline{A \wedge B}\ ab}{A}\wedge E_1 \quad \lambda a \rightarrow \dfrac{}{A}\ a}{B \wedge A}\wedge I}{A \wedge B \longrightarrow B \wedge A}\longrightarrow I}{}$$

The Agda/Alfa environment certainly represents the basic paradigm of natural deduction as typed λ-calculus very faithfully. On the other hand, its general approach has to face the standard issues of scaling up to larger applications: how to incorporate advanced proof tools into its functional programming presentation of formal proofs (Agda), and how to draw large inference trees (Alfa).

ProveEasy versus bidirectional reasoning

ProveEasy [Burstall, 1998] is a small teaching tools for interactive composition of linear textual representations of plain natural deduction proofs. The system is generic in the sense that new rules may be added at any time (by writing functions in the Tcl programming language, involving regular expression matching).

While the basic concepts of ProveEasy are inspired by the tradition of λ-calculus and type theory, its primary format observes the most basic principle of "declarative" proof texts (cf. §4.2.3) by including intermediate propositions explicitly in the text (rules and some instantiations have to be given as well).

Here is a typical proof text produced by ProveEasy. While this form may be entered directly by hand, it is usually composed interactively by pointing at appropriate rules to be applied in the next step.

```
1     . Show  {a & b} -> {b & a}   by  showImp
11.1  . . Given a & b
11    . . Show  b & a   by  givenAnd 11.1
111.1 . . . Given a
111.2 . . . Given b
111   . . . Show  b & a   by  showAnd
1111  . . . . Show  b   by  given
1112  . . . . Show  a   by  given
    QED
```

We may easily reproduce this format in Isabelle/Isar as follows.

lemma $A \wedge B \longrightarrow B \wedge A$
proof (*rule impI*)
  assume *ab*: $A \wedge B$
  show $B \wedge A$
  proof (*rule conjE* [*OF ab*])
    assume A
    assume B
    show $B \wedge A$
    proof (*rule conjI*)
      show B **by** *assumption*
      show A **by** *assumption*
    qed
  qed
qed

The natural deduction format of ProveEasy is restricted to pure backwards reasoning, as represented by the above slightly formalistic use of plain **assume** and **show** without ever using forward-chaining (via **then**). Furthermore, ProveEasy does not admit auxiliary facts to be established separately (cf. **have** in Isar), this has to be achieved indirectly by having assumptions emerge just in the right way to be used later on. Assumptions are indeed the only facts that may be referenced, as may be seen from the format of labels with a dot given in the ProveEasy text above. In contrast, Isar allows *any* local result to be used later on, even those established by **show** (this results in non-linear reasoning with DAG-shaped internal structure, cf. the Knaster-Tarski example in §1.5).

ProveEasy follows a systematic scheme of complete labeling of intermediate lines and facts, essentially path specifications of the underlying tree structure.

[Lamport, 1994] proposes a similar format of names, which includes some additional notational devices to address the typical proliferation of redundant labels emerging from this technique.

We argue that such complete path specifiers are not quite appropriate in realistic applications. First of all, the tree structure of common proof texts is mostly linear anyway, with only a few forks (typically caused by case analysis or induction). This is the deeper reason why labels in the above ProveEasy text consist of many 1's, but very few 2's. As may observed from the accompanied Isar version, only very few such names are actually used later on. The demand for labeled facts may be reduced even further by proper use of forward-chaining, instead of insisting on strict backward reasoning. With this basic tuning we already arrive at a much smoother Isar version (cf. §4.2.3).

lemma $A \wedge B \longrightarrow B \wedge A$
proof (*rule impI*)
  assume $A \wedge B$
  then show $B \wedge A$
  proof (*rule conjE*)
    assume B **and** A
    then show $B \wedge A$
      by (*rule conjI*)
  qed
qed

Here the primitive composition [*OF ab*] has already been covered by (*rule conjE*) used with the chained fact of $\vdash A \wedge B$; likewise have we have incorporated the assumption steps into (*rule conjI*) as well (after adjusting the order of B and A appropriately).

As already seen in §4.2.3, the actual rule specifications happen to be completely redundant, since the explicit propositions and structural information of the text already provide sufficient clues to determine these behind the scenes.

lemma $A \wedge B \longrightarrow B \wedge A$
proof
  assume $A \wedge B$
  then show $B \wedge A$
  proof
    assume B **and** A
    then show $B \wedge A$ **..**
  qed
qed

We see that immediate forward chaining of existing facts is an important ingredient to streamline natural deduction reasoning, reducing the formal noise of labeled facts.

Here are some further variants that illustrate Isar's liberal approach to mixed forward and backward reasoning; depending on the structural details of the

proof we may be occasionally forced to name facts or standard rules explicitly, though.

lemma $A \wedge B \longrightarrow B \wedge A$
proof
  assume $A \wedge B$
  show $B \wedge A$
  proof
    show B **by** (*rule* $conjD_2$)
    show A **by** (*rule* $conjD_1$)
  qed
qed

lemma $A \wedge B \longrightarrow B \wedge A$
proof
  assume ab: $A \wedge B$
  show $B \wedge A$
  proof
    from ab **show** B **..**
    from ab **show** A **..**
  qed
qed

lemma $A \wedge B \longrightarrow B \wedge A$
proof
  assume $A \wedge B$
  thus $B \wedge A$
  proof
    assume A **and** B
    show *?thesis* **..**
  qed
qed

lemma $A \wedge B \longrightarrow B \wedge A$
proof
  assume ab: $A \wedge B$
  from ab **have** b: B **..**
  from ab **have** a: A **..**
  from b a **show** $B \wedge A$ **..**
qed

As may be observed in the last version above, an extremely forward style of reasoning tends to demand many explicitly named local facts; on the other hand standard rules of inference normally do not need to be named, as each line needs to be closed separately, leaving little choice for the rules to be applied.

Nevertheless, name references may be easily reduced via the most basic derived

Isar commands involving **then** (cf. §3.3.3); **with** is particularly useful in such situations, as it uses the current facts together with earlier ones. Here is a tuned version of that proof.

lemma $A \wedge B \longrightarrow B \wedge A$
proof
  assume *ab*: $A \wedge B$
  hence *b*: B **..**
  from *ab* **have** A **..**
  with *b* **show** $B \wedge A$ **..**
qed

So just by a few "peephole optimizations" we have been able to reduce the total number of name occurrences (both defined and referenced) from 7 to 4; this is the typical rate achieved in real applications as well. Isar also provides further infrastructure beyond basic natural deduction (see chapter 6) to reduce the need for labeled facts even more in large-scale applications (see §6.4.3).

Incidently, the strictly backward natural deduction presentation of ProveEasy [Burstall, 1998] is complemented by the mostly dual one of structured forward reasoning in [Hofstadter, 1979]. Using the latter format, our present example looks like this (cf. [Hofstadter, 1979, chapter VII, p. 184]).

[	push
$<A \wedge B>$	premise
A	separation
B	separation
$<B \wedge A>$	joining
]	pop
$<A \wedge B> \supset <B \wedge A>$	fantasy rule

Hofstadter calls the "[...]" form "phantasy mode", where facts may be locally invented to be discharged later on. The (formal) proof format of [Tutch] [Abel *et al.*, 2001] happens to be almost the same for this example.

```
proof andComm: A & B => B & A =
begin
  [ A & B;            %assumption
    A;
    B;
    B & A ];          %conclusion
  A & B => B & A
end;
```

This kind of forward reasoning may be easily reproduced in Isabelle/Isar via raw blocks (see also §5.2), although we need to name some facts and rules explicitly.

{
  assume $A \wedge B$

have a: A **by** (*rule* $conjD_1$)
have b: B **by** (*rule* $conjD_2$)
have $B \wedge A$ **by** (*rule* *conjI* [*OF* b a])
}
hence $A \wedge B \longrightarrow B \wedge A$ **..**

We see that Isabelle/Isar is able to cover the whole range from purely backwards to purely forwards reasoning from one end to the other, including any conceivable intermediate arrangement as well.

Mizar

Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] has pioneered formal proof construction according to general guidelines of established mathematical practice. The system is tied to classical first-order logic, with a formulation of typed set-theory for actual applications. Consequently, both its internal foundations as well as its primary user experience are farther removed from the pure intuitionistic look and feel of the systems we have considered so far, including the Isabelle/Pure framework of Isar (§2.2).

Mizar provides two main mechanisms of formal proof checking (as implemented in its "verifier"): proof outlining with step-wise refinement according to basic first-order principles, and terminal solving of left-over problems by a builtin notion of "obviousness" [Rudnicki, 1987]. Despite being inherently classical, the outlining mechanisms of Mizar may be used for reasonable representations of plain natural deduction proofs as well.

Here is an attempt to emulate our preferred version of the running example of $A \wedge B \longrightarrow B \wedge A$ in Mizar. As Mizar takes its first-order foundations very seriously, we have to simulate propositional variables via set membership of unspecified individuals.

```
reserve x, y, A, B for set;

theorem x ∈ A & x ∈ B implies x ∈ B & x ∈ A
proof
  assume a: x ∈ A;
  assume x ∈ B;
  hence x ∈ B;
  thus x ∈ A by a;
end;
```

It is important to note that the above order of assumptions and conclusions is fixed, thus we really do need the explicit naming of fact `a`, in order to be able to use it over a distance. Only the second assumption may get used directly by "linking", using `hence` instead of `thus` in the subsequent step.

Isar generally offers more flexibility in arranging the key elements of a proof body. In particular, assumptions may be permuted and repeated in an arbi-

trary manner; conclusions may be rearranged as well. Thus one may easily arrange the text such that corresponding facts are placed next to each other, in order to clarify the proof structure and enable forward chaining to reduce the need for named references. Certainly, assumptions have to be always introduced *before* the corresponding conclusions. In particular, we may not just state an assumptions where it happens to get used, nested within a proper sub-proof of the corresponding conclusion; this restriction enforces static scoping of assumptions (which correspond to λ-abstractions), and compositional proof processing (i.e. sub-proofs may never affect the meaning of the enclosing text).

It is *not* that easy to represent Mizar proofs directly within the Isar framework, due to fundamental semantical differences of how proof outlines are processed.

First of all, we observe that Mizar's `proof` and `end` do not have any separate meaning, but only serve as delimiters of the proof body. Furthermore, `by` refers to the builtin automatic prover used together with a number of additional facts (Mizar's `then` primitive, which is technically encountered in `hence` as well, would just include the most recent fact into that specification). Also note that Mizar's `thus` actually corresponds to Isar's **show**, while Isar's **thus** would be `hence` in Mizar.[1] See also [Wiedijk, 2000] for a more detailed attempt to relate the basic Mizar and Isar language elements to each other, based on a simplified model of either system.

Apart from such superficial differences, the basic model of processing proof outlines in Mizar is fundamentally different from the way that Isar builds up local contexts within a proof body and solves some goals eventually. In case that there is a main goal at the head of the proof (as encountered here), Mizar's operation may be understood as a structured walk through the remaining problem, as it is transformed step-by-step via a number of outline commands: `assume`, `thus`, `hence`, and further ones corresponding to basic first-order connectives and quantifiers, such as `let`, `take`, `consider`, `given` [Trybulec, 1993] (see also §5.5.1). A few logical connectives are treated implicitly, such as implication and conjunction.

Mizar provides a number of additional concepts to represent common patterns of forward reasoning encountered in mathematics, most notably iterated equality reasoning (see also §6.4.1), as well as "diffuse" reasoning without a goal statement at the head position (using "`now` ... `end`" or "`hereby` ... `end`"). Taking these elements away from Mizar, one would basically arrive at a system that is very close to the goal oriented paradigm of tactical proving, only that the set of "tactics" has been chosen more carefully with readability in mind. This basic observation has been a starting point of the "Mizar mode for HOL" [Harrison, 1996b], and has been worked out further in "Mizar-Light" [Wiedijk, 2001b].

This quasi-operational style of stepwise transformations of a single problem at hand cannot be easily reproduced in Isar, which is slightly more "declarative"

[1] This particular terminology of Mizar is not ideal for linguistic reasons: while `hence` would be technically the same as `then thus`, the latter form had to be suppressed due to its odd reading as quasi-natural language.

in the sense that arbitrary goal refinements may only take place in the very first **proof** step, or when performing **qed**. Within an Isar proof body there is no way to work on pending goals directly (there is not even a fixed focus on a particular one). Results that are meant to refine enclosing goals have to be built up strictly declaratively by giving suitable **assume** and **show** statements in the proof body.

Incidently, the treatment of Mizar's `thesis` versus Isar's *?thesis* illustrates the key difference of structured proof processing quite nicely. In Mizar, `thesis` is a special placeholder for the remaining part of the problem one is currently working at in the present section of a proof body; consequently `thesis` is *dynamically* updated after each main step. A trace of the course of value of `thesis` in the above Mizar example may be given as follows.

```
theorem x ∈ A & x ∈ B implies x ∈ B & x ∈ A
proof                  — thesis = x ∈ A & x ∈ B implies x ∈ B & x ∈ A
  assume a: x ∈ A;     — thesis = x ∈ B implies x ∈ B & x ∈ A
  assume x ∈ B;        — thesis = x ∈ B & x ∈ A
  hence x ∈ B;         — thesis = x ∈ A
  thus x ∈ A by a;     — thesis = -
end;
```

In contrast, Isar's *?thesis* is just another term abbreviation that happens to be bound automatically whenever a new claim is stated in the text (cf. §3.2.3). Thus it always refers *statically* to the head of the present proof. Once that the initial goal has been refined in a non-monotonic manner, *?thesis* becomes useless for the current piece of proof text. Updating *?thesis* dynamically as in Mizar would quickly lead to unreadable proofs, as initial goal refinements may involve just any Isar proof method. On the other hand, Mizar's dynamic behavior does not cause any real problems in practice, since the basic transformations available here are limited to a few principles from classical first-order logic that are relatively easy to oversee.

We finally give a try at emulating the present Mizar example in Isar. Recalling one of the more or less canonical Isar versions already encountered before, we see that we require additional nesting of sub-proofs, in order to be able to enter the logical structure of the problem.

lemma $A \wedge B \longrightarrow B \wedge A$
proof
  assume $A \wedge B$
  thus $B \wedge A$
  proof
    assume B **and** A
    thus *?thesis* **..**
  qed
qed

Mizar usually requires less structural overhead to dig into first-order proof problems. On the other hand, this advantage is strictly limited to pure logic. In

contrast, the explicit goal refinements in Isar (via initial or terminal method specifications) may be just anything, ranging from domain-specific introduction and elimination rules declared by the user, to arbitrary automated proof tools. In fact, our theory of intuitionistic first-order logic has been declared as such a "domain-specific" application in the first place.

The gain of flexibility of the Isar framework pays off even more in "realistic" applications of formal logic (e.g. chapter 8, chapter 9, chapter 10). Certainly, concrete applications demand some further infrastructure beyond plain natural deduction; this is easy to achieve on top of the existing Isar framework (see also chapter 5 and chapter 6).

Using advanced Isar techniques to be introduced later on (see §5.3) we may easily turn the tide again in favor of Isar, even for this primitive example.

lemma $A \wedge B \longrightarrow B \wedge A$
proof
  assume $A \wedge B$
  then obtain B **and** A **..**
  thus $B \wedge A$ **..**
qed

Part II

Techniques

Chapter 5

Advanced natural deduction

We explore a broad range of "advanced" natural deduction techniques in Isar. First of all, this includes a systematic exposition of the capabilities of the existing language framework introduced so far, pointing out its practical virtues as opposed to pure λ-calculus notions of formal proof. Furthermore we introduce additional derived concepts, notably generalized elimination as a first-class proof context element, and specific support for common schemes of proof by cases and induction. Any of these techniques turn out as indispensable means to support scalable applications.

5.1 Introduction

Natural deduction has been introduced by [Gentzen, 1935] as a formalism to represent the way that mathematicians perform proof *in principle*. Modern expositions usually explain natural deduction in terms of typed λ-calculus, e.g. see [Thompson, 1991] or [Barendregt and Geuvers, 2001]. This provides a viable formal basis for both theoretical studies and concrete implementations, but it does not immediately offer means for human-readable presentations of formal proof texts. Existing mathematical practice does not quite resemble the pure λ-calculus style of reasoning.

We have already explored some aspects of textual representation of basic natural deduction elements earlier (chapter 4), considering both the Isar view and several other approaches. The Isar proof texts encountered there could be related to the most basic concepts of λ-calculus, namely abstraction for context elements **fix** and **assume**, and application (or general composition) for **show** and rule applications involved in **proof** and **qed** steps. A few derived elements of λ-calculus have already been encountered as well, notably various versions of let-expressions covered by **have**, **note**, and **let**. A modified view on application has been indicated by **then**.

Subsequently, we shall provide a systematic exposition of further elements of natural deduction available in Isar. All of these may be expressed on top of the existing language framework (chapter 3), and would correspond to equally "redundant" additions to the plain λ-calculus view of reasoning. Nevertheless, the resulting Isar proof patterns turn out to be indispensable prerequisites for advanced applications (e.g. chapter 8, chapter 9, chapter 10). Even very simple applications like the Knaster-Tarski Theorem given in §1.5 already benefit greatly from such derived elements.

The general lesson to be learned here is that the subtle task of composing human-readable proof texts needs to be accommodated by a considerable diversity of the formal language. Despite our general aim to keep the very core of the Isar language small, its highly compositional nature results in a rich environment of meaningful proof patterns. This principle holds both for natural deduction proper to be discussed here, as well as its light-weight cousin of "calculational reasoning" (see chapter 6).

The following particular techniques will be explored in the present exposition of advanced natural deduction.

1. Various basic techniques that are already inherently present in the core proof language (chapter 3), but have not been included in the discussion of basic natural deduction so far (chapter 4).

   Speaking again in terms of λ-calculus, this includes "non-standard" concepts like general (cascaded) context elements (see §5.2.1), incremental let-expressions (see §5.2.2), modified application and composition (see §5.2.3), stand-alone parentheses (see §5.2.4), and internalized proof texts in the form of meta-level rule statements (see §5.2.5).

2. Support for generalized eliminations via the derived **obtain** element (see §5.3). This basically amounts to existential quantification at the level of Isar proof texts, or "conservative extensions" of local proof contexts.

   Instead of having context elements **fix** and **assume** emerge implicitly as the result of previous backward reasoning, **obtain** allows to *prove* that parameters and assumptions may be introduced at a certain point (independent of any goals). This principle admits a large number of useful patterns, with considerable elimination of formal noise.

3. Specific infrastructure for proof by cases and induction (see §5.4) that scales up well in practice.

   Depending only on a few additional proof methods and attributes, the existing **case** command (§3.3.1) is turned into a viable tool to introduce large context elements into proof texts succinctly, corresponding to canonical rules of inductive sets or types.

Interestingly, the "advanced" issues covered here will mostly revolve around static proof contexts rather than dynamic goal configurations. This observation

marks a distinctive difference of structured proof techniques versus existing approaches of goal-oriented tactical theorem proving, like in traditional Isabelle [Paulson and Nipkow, 1994], the HOL system [Gordon and Melham, 1993], or Coq [Barras *et al.*, 1999]). The calculational reasoning techniques of chapter 6 will drive this view to its ultimate consequence, arriving at a proof style that is essentially devoid of goals altogether.

5.2 Basic techniques

5.2.1 General context elements

The Isar framework provides the two fundamental context primitives **fix** and **assm** (cf. §3.2.1). Speaking in terms of λ-calculus both essentially correspond to abstraction: **fix** abstracts over terms (with syntactic types) and **assm** over facts (or rather internal proof terms). A proof text involving "**fix** $\vec{x}$ **assm** $\vec{H}$" corresponds directly to an internal context of a proposition presented in HHF normal form $\bigwedge \vec{x}.\ \vec{H} \Longrightarrow H$ (cf. §2.4.1).

As we shall elaborate below, canonical equivalence transformations of such formula may be performed on Isar proof texts as well, e.g. α-conversion of parameters, permuting and repeating premises, and commuting parameters with premises (according to the law $\vdash (P \Longrightarrow (\bigwedge x.\ Q\ x)) \equiv (\bigwedge x.\ P \Longrightarrow Q\ x)$).

The **assm** (§3.2.1) primitive is not directly available in Isar proof texts, but is intended to implement user-level elements accordingly, such as **assume**, **presume**, **def**, and **case** (cf. §3.3.1). Later on we will also introduce the derived **obtain** element (see §5.3), and cover advanced uses of **case** (see §5.4). In contrast, raw **fix** is slightly more degenerate as an individual concept, since it does not involve any special treatment at discharge time. On the other hand, derived context elements may refer to **fix** and **assm** simultaneously, like **def** or **obtain**.

Fixed variables

Variables introduced via **fix** refer to local objects that are purely syntactic: when exporting results such elements may be generalized according to the canonical $\bigwedge$ introduction rule (cf. §2.2). No additional hypotheses wrt. typing of variables are imposed here, because the underlying framework inherently assumes types are always inhabited (see also §8.6.1 for the analogous situation in the HOL object-logic). Non-dependent expressions of $\bigwedge$ may be immediately simplified according to the law $\vdash (\bigwedge x.\ P) \equiv P$.

Portions of Isar proof texts involving **fix** are α-convertible, just like the corresponding $\bigwedge$ binder expressions of the underlying logical framework (§2.2). Consider the following trivial example.

have $\forall x.\ P\ x$
proof
  fix x
  show $P\ x$ ⟨*proof*⟩
qed

have $\forall x.\ P\ x$
proof
  fix u
  show $P\ u$ ⟨*proof*⟩
qed

In fact, the result of a proof body needs to conform to a pending goal only up to higher-order unification (§3.2.3). Thus the text may actually cover a more general statement, if it happens to be provable at that level of generality.

have $\forall x.\ P\ (f\ x)$
proof
  fix y
  show $P\ y$ ⟨*proof*⟩
qed

We see that Isar proof texts are not bound to the accidental formulation of goal statements, but may raise the level of abstraction at will. Most existing proof systems are directly focused on particular goal statements instead, cf. the `Intros` element of Coq [Barras *et al.*, 1999]), for example. Interestingly, even `let/assume` in Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] are essentially based on the same procedural paradigm (see also §4.2.4 and §5.5.1) as the tactical view of Coq.

The following Isar example exploits the idea of generalized proof bodies in order to re-use an existing proof a second time for a symmetric argument.

lemma $(A \vee B) = (B \vee A)$
proof
  fix $X\ Y$
    — general propositions X, Y may get instantiated later on
    — either as A, B or B, A
  assume $X \vee Y$
  thus $Y \vee X$
  proof
    assume X
    thus *?thesis* **..**
  next
    assume Y
    thus *?thesis* **..**
  qed
    — first result application

thus $Y \vee X$ **.**
— second (symmetric) result application
qed

Here we have exploited another intrinsic virtue of Isar proofs, namely "cascading" of contexts. Having exported a result (cf. the first "**thus** $Y \vee X$ ⟨*proof*⟩" above) does not yet invalidate the existing context built up so far, including any kind of local proof items like auxiliary facts, term abbreviations, or proper logical context elements. The second "**thus** $Y \vee X$ **.**" applies the very same result $\vdash \bigwedge X\ Y.\ X \vee Y \Longrightarrow Y \vee X$. In general, later results may involve longer $\bigwedge/\Longrightarrow$ prefixes due to additional context commands issued intermediately.

This incremental behavior is an immediate consequence of the way that the Isar/VM interpreter manages the corresponding environments of the static proof configuration of *context* (cf. §3.2.3). Speaking in terms of plain λ-calculus, certain parts of nested abstractions (and other binder elements) may be shared among several expressions, resulting in slightly less formalistic proof texts by preferring linearized arrangements over strongly nested ones.

Strong assumptions

Unquestionably, **assume** is the most fundamental proof context element. It introduces a "strong" assumption in the sense that exported results need to unify against corresponding premises of an enclosing goal. Thus finished proof fragments of **assume**/**show** essentially provide a balanced textual focus on a particular open problem, covering both assumptions and conclusions. This allows proof bodies to be commuted in many situations where conclusions alone would cause ambiguities. Consider this basic example of $\vee$ elimination.

assume $A \vee B$
hence C
proof
  assume B — second case
  thus C ⟨*proof*⟩
next
  assume A — first case
  thus C ⟨*proof*⟩
qed

Furthermore, strong assumptions may be introduced in any order or even repeatedly, without changing the behavior of Isar proof processing (cf. §3.2.3).

assume $A \wedge B$
hence $B \wedge A$
proof
  assume A — (unused)
  assume B — (unused)
  assume B **and** A

```
  thus ?thesis ..
qed
```

Note that this liberal treatment of assumptions is quite important in practice to tune proof texts according to the most natural flow of information, both for interactive development and improved readability of the final text. In particular, properly arranged facts often avoid explicit references to facts and rules (cf. the discussion in §4.2.4, see also §5.2.3).

On the other hand, assumptions and corresponding goal statements must not be swapped. In the subsequent example, **assume** is introduced properly *before* its related **show** (demanding an explicit label).

```
have A ⟶ B
proof
  assume a: A
  show B
  proof −
    from a
    show ?thesis ⟨proof⟩
  qed
qed
```

In contrast, the subsequent attempt of introducing assumptions "dynamically" when required does *not* work out, since it violates scoping of logical context elements (abstractions).

```
have A ⟶ B
proof
  show B
  proof −
    assume A
      — illegal "dynamic" assumption
    thus ?thesis
      ⋮
```

The latter version could spare us an explicit reference to the previous fact a, but it is unacceptable for several reasons. For example, it would break modularity of Isar proof checking: the particular context introduced within the body of a sub-proof would change the meaning of the main proof. Also note that the correct use of **assume** before **show** needs to impose the resulting hypothesis on the result independently of its actual use in the sub-proof. Generally speaking, the Isar/VM interpreter (§3.2.3) implements a discipline of "static scoping" of proof contexts in order to get these subtle details right. Such fine points are just too easily overlooked in "real world" implementations of interpreted languages, although it has been a well-known issue of proper programming language semantics for several decades. The initial error of dynamic scoping in LISP interpreters of the late 1950's should have been overcome now (at least in theory) [McCarthy, 1960].

Other context elements

The **presume** element provides "weak" assumptions: unlike **assume** the discharged hypotheses are *not* solved against any goal premises. Thus former presumptions are left as new sub-problems to be solved later on. So **presume** essentially defers sub-proofs according to a logical "cut" rule.

Just consider the following simple example, involving the rule $r = \vdash A \Longrightarrow C$.

have C
proof (*rule* r)
  presume B
  thus A ⟨*proof*⟩
next
  show B ⟨*proof*⟩
qed

In practice, **presume** turns out to be most useful in interactive development where portions of a proof may be temporarily deferred, or to debug failed applications of **assume**/**show** due to faulty assumptions. In such situations **assume** may be temporarily replaced by **presume** to inspect partially applied results of **show**, with pending subgoals corresponding to previous presumptions.

Note that exporting results from a context of weak assumptions does not involve any special treatment of premises in the enclosing goal context (cf. §3.3.1). Thus the effect is essentially the same as in applications of non-atomic rule statements.

have C
proof (*rule* r)
  show $B \Longrightarrow A$ ⟨*proof*⟩
  show B ⟨*proof*⟩
qed

The **def** element essentially provides an abbreviation for "**fix** x **assume** $x \equiv t$" where the discharged (and generalized) equation is automatically disposed of via reflexivity $\vdash t \equiv t$. So **def** performs a *definitional extension* of the present proof context (cf. §2.3 for a similar principle for the theory level). The **def** element is best studied within a raw proof block, see also §5.2.4. Another characterization is given in §5.3.3, reducing basic **def** to more the sophisticated **obtain** element of generalized elimination (which corresponds to general *conservative extensions*).

Interestingly, **let** is much more relevant in practice. Unlike **def** it is not a logical context element, but merely an extra-logical device of term abbreviations (§3.2.3 and §3.4.1). Its very power stems from this arrangement, including conveniences such as higher-order matching or Hindley-Milner polymorphism (§3.4.3) without requiring the underlying logical framework to take care of any of these additional concepts.

The **case** element provides a generic interface to invoke named context segments of the form "**fix** $\vec{x}$ **assume** $\vec{\varphi}$". Cases typically emerge from canonical proof

patterns that have been initially applied to the present goal configuration, see also the *cases* and *induct* methods in §5.4. Non-atomic claims also give rise to the named case of *antecedent* referring to the pending rule context (see §5.2.5).

5.2.2 Local facts and goals

In principle, the **show** goal element of Isar (§3.2.1) is already sufficient to support natural deduction proofs, there is no particular need for the **have** version. Speaking in terms of pure λ-calculus, **show** closely corresponds to application (of a local result to a pending goal). Limiting local results to immediately pending goals turns out as slightly impractical, though (cf. the discussion of ProveEasy [Burstall, 1998] in §4.2.4).

Isar's **have** element does not attempt to refine any goal yet, but merely exhibits the result to the present proof context. In a sense, **have** removes the "tension" from **show** to fit into an enclosing proof problem. The behavior of **have** resembles let-expressions in λ-calculus, it binds a local fact to be used eventually in the subsequent body. Note that Isabelle/Isar even implements the well-known polymorphic version of let-binding according to Hindley-Milner (cf. §3.4.3).

The effect of "cascaded contexts" already observed in §5.2.1 holds for local facts as well. Already proven facts (both from **show** or **have**) may be re-used later on without further ado. Isar essentially uses general DAG-shaped organization of local results rather than pure tree structure. Recall the Knaster-Tarski Theorem (cf. §1.5) for a somewhat realistic DAG-shaped proof. Here the corresponding primitive representation duplicates the primitive proof of the shared fact $ge = \vdash f\ (\bigcap\{u.\ f\ u \subseteq u\}) \subseteq \bigcap\{u.\ f\ u \subseteq u\}$ due to internal β-normalization.

The **show** element solves both an enclosing goal and exhibits the result locally (while **have** only does the latter). Consequently, multiple goals may be covered in a sequential manner, by re-using previously proven facts as illustrated below.

have $A \wedge B$
proof
  show A ⟨*proof*⟩
  from *this* **show** B ⟨*proof*⟩
qed

Here we have included the first conjunct in the proof of the second one, analogous to the rule $\vdash A \Longrightarrow (A \Longrightarrow B) \Longrightarrow A \wedge B$. The order of sub-proofs may be changed as well, since there is no fixed goal focus in Isar (§3.2.3). The following pattern corresponds to the symmetric rule $\vdash B \Longrightarrow (B \Longrightarrow A) \Longrightarrow A \wedge B$.

have $A \wedge B$
proof
  show B ⟨*proof*⟩
  from *this* **show** A ⟨*proof*⟩
qed

Both **proof** steps above have used the rule $\vdash A \Longrightarrow B \Longrightarrow A \wedge B$ invariably. We see that the Isar infrastructure of cascaded local facts "enhances" plain natural deduction rules in a casual manner.

5.2.3 Mixed forward and backward reasoning

We have already explored variations on forward versus backward reasoning in §4.2.4. Isar's flexibility in this respect (by virtue of the **then** element) turns out as an important ingredient to achieve readable proof texts after all. The general principle observed here may be illustrated by the following model of the general situation. Consider the following natural deduction rule:

$$\frac{a_1\colon A_1 \quad a_2\colon A_2 \quad b_1\colon B_1 \quad b_2\colon B_2 \quad b_3\colon B_3}{C}\ (r)$$

Here the premises of r are divided into a prefix of $a_i\colon A_i$ that is most appropriately filled in by existing facts (e.g. previous assumptions), and a suffix of $b_j\colon B_j$ that are better solved by separate sub-proofs later (typically the latter involves increasingly complex statements that require separate universal parameters and assumptions, due to nested $\bigwedge$ and $\Longrightarrow$ in the rule). Most common natural deduction rules follow this basic arrangement, e.g. consider $\exists$ elimination $\vdash \exists x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow C) \Longrightarrow C$ with exactly one "A" and one "B" premise. Rules like $\wedge$ introduction $\vdash A \Longrightarrow B \Longrightarrow A \wedge B$ are more regularly shaped and admit several valid divisions into prefix and suffix parts, depending on the particular situation in the proof text at hand.

This split view on rule r gives rise to the following mixed forward-backward proof pattern: "A" premises are established via **have** in the preceding context, while the "B" ones are covered in the body of the main claim via **show**.

have $a_1\colon A_1$ ⟨*proof*⟩

⋮

have $a_2\colon A_2$ ⟨*proof*⟩

⋮

from a_1 **and** a_2
have C
proof (*rule* r)
  show $b_1\colon B_1$ ⟨*proof*⟩

  ⋮

  show $b_2\colon B_2$ ⟨*proof*⟩

  ⋮

  show $b_3\colon B_3$ ⟨*proof*⟩
qed

This arrangement distributes the corresponding sub-proofs over the Isar text nicely, with the "A" given in advance and the "B" ones within the body. Recall that the basic *rule* method demands chained facts to be given in proper order (§3.3.2), the **from** line above needs to take care of this. We do not attempt to build ad-hoc costly permutations into basic steps, any serious proof search is better left to explicit automated proof methods (like *blast* in classical Isabelle/HOL, see §7.3).

When conducting single natural deduction proof steps in practice, the canonical order of premises to be filled in beforehand often coincides with the most natural arrangement of the corresponding pieces of proof text. Standard natural deduction rules already tend to be formulated that way. Furthermore, there is normally that split into "A" and "B" parts as a prefix and suffix of rule premises, respectively. In rare circumstances, fact positions may be skipped using the global fact "-", which refers to to $\vdash \bigwedge A.\ A \Longrightarrow A$. This is illustrated by the following synthetic example.

have *b*: B ⟨*proof*⟩

⋮

from - *b* **have** $A \wedge B$
proof (*rule conjI*)
  show A ⟨*proof*⟩
qed

Note that we need to specify the $\wedge$ introduction rule explicitly, since the dummy fact "-" does not provide any structural clue about the intended proof step. Plenty of non-sensical eliminations would be tried before any introduction.

In fact, the seamless way of single step reasoning in Isar heavily depends on proper indication of the use of existing facts (cf. the related discussion in §4.2.4). Otherwise, rather verbose method specifications need to be given, cluttering the proof text unnecessarily. A good balance of facts versus methods turns out as a key factor to achieve readable proof texts. This is the deeper reason why the (theoretically) redundant **then** modifier is so important in Isar.

In conclusion we present a simple example of shifting the balance of indicating facts versus proof methods back and forth. Assume that $ab = \vdash A \Longrightarrow B$ and $a = \vdash A$ are available in the present context.

have B **by** (*rule ab*) (*rule a*)
  — two methods (initial and terminal)

from a **have** B **by** (*rule ab*)
  — fact chained towards method

from ab **and** a **have** B **.**
  — two facts applied immediately

The second form is most frequently encountered in practice. This scheme may

be generalized to any number of facts and arbitrary complex proof tools (see also §7.3). Then it nicely achieves an indication of the "relevance of facts" in the text, while leaving method specifications uncluttered from additional arguments (see the related discussion in §7.5.2).

5.2.4 Raw proof blocks

Block structure is readily available in Isar proofs, but is normally not made explicit in the text. Sub-proofs implicitly live within their own local context (§3.2.3), without requiring separate parentheses given by the user. Even more, the **next** command allows to "jump" blocks without requiring separate close/open specifications. Nevertheless, explicit block structure is occasionally quite useful as well. Isar provides the "**{**" and "**}**" elements to delimit proof blocks separately (§3.2.3); this essentially acts like a headless sub-proof (lacking the initial claim) where facts may freely float into "**{**" and out of "**}**".

Forward composition of rules

The result of a proof block is its final fact exported into the enclosing context. Local assumptions etc. are discharged accordingly. Thus we may prove rule statements in a forward fashion, as illustrated below by the well-known propositional rules of K and S.

{
  assume A **and** B
  have A **.**
} **note** $K = this \ \ \text{—} \vdash A \Longrightarrow B \Longrightarrow A$

{
  assume x: $A \Longrightarrow B \Longrightarrow C$
  assume y: $A \Longrightarrow B$
  assume z: A
  have C
  proof (*rule* x)
    show A **by** (*rule* z)
    show B
    proof (*rule* y)
      show A **by** (*rule* z)
    qed
  qed
} **note** $S = this \ \ \text{—} \vdash (A \Longrightarrow B \Longrightarrow C) \Longrightarrow (A \Longrightarrow B) \Longrightarrow A \Longrightarrow C$

The general idea is to spell out the final result near the beginning of the block via **assume** and **have**. The result needs to be named afterwards, since any local bindings would be invisible outside of the block. The following variations on S rearrange the course of reasoning internally. Note that the parts contributing

to the final result (assumptions and the last fact) may not be reordered without affecting the exported rule itself.

{
assume x: $A \Longrightarrow B \Longrightarrow C$
assume y: $A \Longrightarrow B$
assume z: A
from y **and** z **have** B **.**
with x **and** z **have** C **.**
} **note** $S = this \ \text{—} \vdash (A \Longrightarrow B \Longrightarrow C) \Longrightarrow (A \Longrightarrow B) \Longrightarrow A \Longrightarrow C$

{
assume x: $A \Longrightarrow B \Longrightarrow C$
assume $A \Longrightarrow B$ **and** z: A **hence** B **.**
with x **and** z **have** C **.**
} **note** $S = this \ \text{—} \vdash (A \Longrightarrow B \Longrightarrow C) \Longrightarrow (A \Longrightarrow B) \Longrightarrow A \Longrightarrow C$

As more and more intermediate statements are introduced in the body above it becomes increasingly difficult to determine the final result from the given text. Just like many other useful concepts of Isar, some taste and discernment is required of the writer, lest the text become incomprehensible for the reader; recall our general principle of liberality (§1.3). Note that rules may be established via plain backward reasoning as well (see also §5.2.5).

Blocks as a logic laboratory

Another virtue of raw proof blocks is exhibited in experimentation and teaching of natural deduction proof composition. Due to the absence of an immediate goal context, the behavior of logical declarations may be studied in isolation. This technique basically amounts to some kind of "formal logic laboratory". In particular, the characteristics of **assume**/**presume** and **def** may be expressed within Isar itself quite succinctly as follows.

{
assume A
have C ⟨*proof*⟩
} — $this = \vdash A \Longrightarrow C$

{
presume A
have C ⟨*proof*⟩
} — $this = \vdash A \Longrightarrow C$

{
def $x \equiv t$
have $P\ x$ ⟨*proof*⟩
} — $this = \vdash P\ t$

Note that **assume** and **presume** are really the same in pure forward reasoning, they only differ when exporting results into a goal context (cf. §3.3.1). Moreover, the intra-logical nature of **def** results in a new local object x that is treated as opaque by default, until the fact $\vdash x \equiv t$ is unfolded explicitly. This amounts to ad-hoc abstraction of a concrete expression at hand, which already is the main reason why **def** may get used occasionally instead of the slightly more convenient **let** element for term abbreviations (§3.2.3 and §3.4.1).

Note that **let** turns out as more useful in practice since it merely is a "formal illusion" that does not have any impact on the logical context at all. As illustrated below nothing needs to happen at discharge time, the abbreviations are fully expanded before passing the input down to the logical inference machinery.

{
  let *?x* = *t*
  have *P ?x* — *?thesis* = *P t*
    ⟨*proof*⟩ — *this* = ⊢ *P t*
} — *this* = ⊢ *P t*

The full power of **let** is unleashed by higher-order matching (cf. §3.4.1), which allows to analyze the structure of statements in an extra-logical fashion.

Proof inversion patterns

Tactical theorem proving is strongly biased towards backwards reasoning, where an initial claim is refined consecutively until a finished state is achieved. This mode of operation is particularly subtle in conjunction with automated methods that "simplify" goals (like *simp* or *auto* in Isabelle/HOL, see §7.3).

Isar does not pose any restrictions on methods used in initial proof steps. So one might come up with the following pattern of automated backwards reasoning in structured Isar proofs as well.

have *C*
proof *auto* — automated initial step (generally a bad idea)
  fix *x*
  assume *A x*
  show *D* ⟨*proof*⟩
qed

From the operational viewpoint, this proof text corresponds to a tactic script that would usually be considered as slightly "unstable" according to Isabelle folklore. The problem is that the behavior of the initial *auto* step is very sensitive to changes of the collection of global rule declarations of the background theory (see also §7.3 and §7.4). The refined situation needs to be covered in the Isar proof body via explicit parameters and propositions given in the text. Automated tools usually become stronger as the library evolves over the years, which might cause slightly different local problems to emerge eventually, demanding the writer to adapt the original formulation accordingly.

have *C*
proof *auto* — automated initial step (generally a bad idea)
  fix *x* **and** *y*
  assume *A′ x y* **and** *B x y*
  show *D′* ⟨*proof*⟩
qed

Is a rather bad idea to let arbitrary automated tools determine the decomposition of problems in structured proof texts. Unstructured scripts are less sensitive to such problems, as explicit statements are avoided as much as possible in the first place (rendering the script unreadable, of course).

The following simple technique of "inverted proofs" achieves more robust proof texts that are invariant under *monotonic* changes of of automated tools. Of course, the inner proof block may still have emerged from a previous experimental phase of automated backwards refinement as seen before.

have *C*
proof −
  {
    fix *x*
    assume *A x*
    hence *D* ⟨*proof*⟩
  }
  thus *?thesis* **by** *auto*
qed

Certainly the final integration phase may still break down if the behavior of *auto* changes in an uncouth manner (which occasionally happens in reality). On the other hand, there is now a significantly higher level of tolerance built into the text. In fact, the above pattern may be considered as a very simple instance of "big-step reasoning", where portions of text are composed loosely and the exported result is finally included in an automated step. See §6.4.3 for the related discussion of "degenerate calculations" in Isar.

5.2.5 Non-atomic statements

The Isar proof language exploits the full potential of higher-order nested natural deduction of the basic logical framework (cf. §2.2): assumption and conclusion statements may be arbitrary meta-level propositions, there is no artificial restriction to atomic ones (i.e. those of the object-logic).

Interestingly, traditional Isabelle tactic scripts [Paulson and Nipkow, 1994] are quite limited in this respect. Special treatment is required for any goal statement with non-atomic premises (operationally similar to the technique of raw proof blocks covered in §5.2.4). Even worse the most basic Isabelle tactics are unable to treat non-atomic facts and premises as expected (most notably `assume_tac`

[Paulson, 2001b]). In a sense, Isabelle's old-style user experience tends to emulate the original HOL tradition of tactical proving [Gordon and Melham, 1993] [Gordon, 2000], rather than staying faithful to its own roots [Paulson, 1989] [Paulson, 1990]. Automated tactics in Isabelle essentially used to share the same problem, but have been enhanced for use in Isabelle/Isar [Wenzel, 2001a] by providing a filter that internalizes meta-level $\bigwedge/\Longrightarrow$ statements into the object-logic via separate $\forall/\longrightarrow$ connectives behind the scenes.

The most basic technique to derive arbitrary rules in Isar is to compose a local proof context that mimics the top-level structure of the statement, claiming the conclusion as a new goal. This reduces the rank of the original problem, while exhibiting its assumptions as local facts to the subsequent proof text. Consider the following simple example.

have $\bigwedge x\ y.\ A\ x\ y \Longrightarrow B\ x\ y \Longrightarrow C\ x\ y$
proof −
  fix x **and** y
  assume a: $A\ x\ y$ **and** b: $B\ x\ y$
  thus $C\ x\ y$ ⟨*proof*⟩
qed

One might consider to address the constituents of the initial rule statement as term abbreviations introduced beforehand, avoiding to repeat all of these (probably large) propositions in the text.

have $\bigwedge x\ y.\ A\ x\ y \Longrightarrow B\ x\ y \Longrightarrow C\ x\ y$
  (**is** $\bigwedge x\ y.\ ?A\ x\ y \Longrightarrow ?B\ x\ y \Longrightarrow ?C\ x\ y$)
proof −
  fix x **and** y
  assume a: $?A\ x\ y$ **and** b: $?B\ x\ y$
  thus $?C\ x\ y$ ⟨*proof*⟩
qed

Alternatively, we may use the standard infrastructure for goal statements provided by the Isar interpreter (§3.2.3), which includes the *antecedent* case representing the original $\bigwedge/\Longrightarrow$ context symbolically, and the *?thesis* abbreviation for the conclusion. Note that the latter is abstracted over any outer universal parameters. Thus we may achieve an almost fully symbolic proof body as follows, where only the local parameters need to be repeated in the text.

have $\bigwedge x\ y.\ A\ x\ y \Longrightarrow B\ x\ y \Longrightarrow C\ x\ y$
proof −
  case *antecedent*
  thus *?thesis* $x\ y$ ⟨*proof*⟩
qed

This form has the minor drawback that "**case** *antecedent*" introduces the full assumption context simultaneously. Thus we may not directly refer to facts assumptions, such as $a = \vdash A\ x\ y$ and $b = \vdash B\ x\ y$ encountered before. Isar

intentionally refrains from low-level operations on lists of theorems (facts and goals are generally treated as opaque).

In common applications of the previous pattern one needs to refer only to few assumptions separately, while all others are covered collectively. There are essentially two different ways to get hold of individual assumptions from the present context, either use **have** with an immediate proof, or repeat relevant assumptions by separate **assume** elements. Recall that Isar proof contexts are invariant wrt. commuted or duplicated entries (cf. §5.2.1).

have $\bigwedge x\ y.\ A\ x\ y \Longrightarrow B\ x\ y \Longrightarrow C\ x\ y$
proof −
  case *antecedent*
    — performs "**fix** x **and** y **assume** $A\ x\ y$ **and** $B\ x\ y$" simultaneously
  have a: $A\ x\ y$ **.** — extracted result
  assume b: $B\ x\ y$ — repeated assumption
  show *?thesis x y* ⟨*proof*⟩
qed

Apart from the basic **have** form above there are further advanced techniques of exploiting the implicit context information provided by **case**, especially in conjunction with **obtain** (see §5.3).

When proving non-atomic statements the initial step need not necessarily be idle. Apart from the "−" method encountered so far, we may as well use plain *rule* steps. Since rule application via higher-order backchaining (§2.4) is monotonic wrt. the internal goal context, the previous techniques of **fix**/**assume** or "**case** *antecedent*" are still applicable. Only the conclusion may require different treatment due to the initial refinement.

The following (synthetic) example illustrates a typical situation where the original goal context is augmented due to the initial **proof** step.

have $\bigwedge x.\ A\ x \Longrightarrow B\ x \Longrightarrow C \longrightarrow D$
proof
  fix x
  assume $A\ x$ **and** $B\ x$ **and** C
  thus D ⟨*proof*⟩
qed

Monotonic rule application like this is desirable in most applications involving non-atomic claims, with the notable exception of induction. Isar provides specific support for induction over rule statements, where the *whole* configuration participates in the recursive reasoning (see also §5.4.5). Here the capabilities of Isabelle tactic scripts would fail altogether, demanding to switch back to object-level $\forall / \longrightarrow$ connectives in order to make the intended "rule" appear as an atomic conclusion only. This would typically demand additional steps to strip connectives later on, causing an excessive dose of formal noise.

5.3 Generalized elimination

Consider the canonical $\exists$ elimination rule of natural deduction:

$$\frac{\exists\, x.\ P\ x \qquad \begin{array}{c}[x,\ P\ x]\\ \vdots\\ C\end{array}}{C}$$

Taking this as a model, the basic idea of *generalized elimination* may be described as follows. At a certain point in a proof, local parameters with additional properties may be introduced such that subsequent results not mentioning these auxiliary parameters are exported without acquiring additional hypotheses.

In other words generalized elimination corresponds to a *conservative extension* of the proof context, where auxiliary parameters and assumptions may be introduced without affecting any self-contained results (cf. §2.3 for a similar concept at the level of theories rather than proofs).

Isar supports generalized elimination by the derived command **obtain**, as introduced below (see §5.3.1 and §5.3.2). This turns out as a very powerful mechanism, both from the basic logical point of view, as well as in actual applications. As one of its most important virtues, **obtain** is able to hide the inherent complexities of elimination rules that involve new local parameters. Thus Isar proof texts may be kept clean from unnecessary formal noise, supporting a plain linear format that is close to the casual style handling existential parameters in common mathematical practice. In order to illustrate the fundamental difference, consider the following two versions of $\exists$ elimination turned into Isar text.

assume $\exists\, x.\ P\ x$
hence C
proof
  fix x **assume** $P\ x$
  thus C ⟨*proof*⟩
qed

assume $\exists\, x.\ P\ x$
then obtain x **where** $P\ x$ **..**
hence C ⟨*proof*⟩

The first version directly mimics the primitive $\exists$ elimination rule, which involves an additional level of nesting and requires an explicit goal statement C (cf. §4.2.2). The second version is slightly more handsome, thanks to a better policy imposed on internal reasoning steps, as we shall see later on.

The **obtain** element supports more liberal *linear* arrangements of proof text, avoiding separate nesting of sub-proofs altogether. Any number of results may be established in the context of **obtain**. The user does not even have to think about these at the point where generalized elimination is performed. So we

do not need to state the ultimate result of C yet, but may just explore the present situation in a forward manner. This results in a style of structured proof composition that is mostly liberated from explicit goals.

assume $\exists\, x.\ P\ x$
then obtain x **where** $P\ x$ **..**

After having sorted out the logical foundations of **obtain** (see §5.3.1), as well as proper support for realistic soundness proofs of the existential claim involved (see §5.3.2), we shall discuss further useful Isar proof patterns of generalized elimination later on (see §5.3.3).

5.3.1 Obtaining contexts

The derived command **obtain** extends the basic syntax of Isar commands (cf. chapter 3) as follows.

obtain $(var^+$ **where**$)^?$ $(name\text{-}atts\text{:})^?$ $prop^+$ (**and** $(name\text{-}atts\text{:})^?$ $prop^+)^*$

The basic logical idea behind **obtain** is expressed by the subsequent (simplified) implementation in a concise manner. We reserve the theorem name *reduction* for internal use. Furthermore we use the generic **assm** primitive (§3.2.1) with the derived rule scheme of *eliminate* as given below.

obtain $\vec{x}$ **where** $q_1\text{:}\ \vec{\varphi}_1$ **and** $\ldots$ **and** $q_n\text{:}\ \vec{\varphi}_n$ $\langle proof\rangle$ $=$
  have *reduction*: $\bigwedge C.\ (\bigwedge \vec{x}.\ \vec{\varphi}_1\ \ldots\ \vec{\varphi}_n \Longrightarrow C) \Longrightarrow C$ $\langle proof\rangle$
  fix $\vec{x}$ **assm** $\ll$*eliminate reduction*$\gg$ $q_1\text{:}\ \vec{\varphi}_1$ **and** $\ldots$ **and** $q_n\text{:}\ \vec{\varphi}_n$

$$\frac{\Gamma \vdash \bigwedge C.\ (\bigwedge \vec{x}.\ \vec{\varphi}_1\ \ldots\ \vec{\varphi}_n \Longrightarrow C) \Longrightarrow C \qquad \Gamma \cup \{\vec{\varphi}_1\ \ldots\ \vec{\varphi}_n\} \vdash \psi}{\Gamma \vdash \psi}\ (\mathit{eliminate})$$

proviso: $\vec{x}$ not free in Γ or ψ

The rule *eliminate* is easily derived within the basic logical framework (§2.2): from $\Gamma \cup \{\vec{\varphi}_1\ \ldots\ \vec{\varphi}_n\} \vdash \psi$ discharge the additional assumptions and generalize over $\vec{x}$ (clearly this will not affect Γ or ψ due to the proviso); then specialize the proposition C of the reduction statement to ψ, and finally apply modus ponens.

We see that **obtain** is somehow dual to plain local statements established via **have**: while "**have** φ $\langle proof\rangle$" means a certain conclusion *holds* in the present context, "**obtain** $\vec{x}$ **where** $\vec{\varphi}$ $\langle proof\rangle$" says that we *may assume* a number of facts involving new local parameters. This observation of duality is related to the fundamental HHF format $\bigwedge \vec{x}.\ \vec{H} \Longrightarrow H$ of logical statements (§2.4), where the left-hand side admits parameters and multiple assumptions, while the right-hand side consists of a single conclusion only.

Another insight on the essence of **obtain** may be gained by looking closely at the *reduction* part. Apparently, this statement expresses "existence" of elements with certain properties: $\bigwedge C.\ (\bigwedge x.\ P\ x \Longrightarrow C) \Longrightarrow C$ coincides with the usual definition of $\exists\, x.\ P\ x$ within a higher-order framework (e.g. see §8.1.2). Likewise, multiple parameters correspond to nested quantifiers and multiple assumptions to conjunction. Without the detour via explicit $\exists$ and $\wedge$ connectives, we may understand the *reduction* statement more abstractly as a conservative extension of the proof context, it explicitly states that self-contained results C may get rid of the temporary assumptions introduced beforehand without acquiring additional hypotheses. This observation is exploited in the *eliminate* rule.

The remaining issue of practical usability of **obtain** is how to perform the proof of *reduction* adequately. Due to its nesting to the left-hand (negative) side of meta-level connectives, the raw statement as given above is slightly awkward to handle directly via basic proof steps. Practical applications really demand some further refinement of the proof obligation encountered here.

5.3.2 Supporting realistic soundness proofs

In order to support realistic proofs of the *reduction* statement of **obtain**, we shall now give more elaborate definition. The basic idea is to break up the raw soundness statement, such that its structure is directly exposed to the Isar proof text, rather than as a primitive proposition. While being technically subtle to design, the resulting proof situation admits a number of quite natural patterns to complete the soundness proof, either in single steps or by automated tools. That additional complexity is hidden from readers of Isar proof texts, while writers may choose to ignore a few superficial details going on internally and merely get acquainted with a number of common reasoning patterns.

For the subsequent full definition of **obtain**, let $\langle fact \rangle$ refer to any previous results indicated for use in forward chaining.

$\langle fact \rangle$ **obtain** $\vec{x}$ **where** q_1: $\vec{\varphi}_1$ **and** ... **and** q_n: $\vec{\varphi}_n$ $\langle proof \rangle$ =
  have *reduction*: $\bigwedge C.\ (\bigwedge \vec{x}.\ \vec{\varphi}_1\ \ldots\ \vec{\varphi}_n \Longrightarrow C) \Longrightarrow C$
  proof *succeed*
    fix *thesis*
    assume *that* [*intro*]: $\bigwedge \vec{x}.\ \vec{\varphi}_1\ \ldots\ \vec{\varphi}_n \Longrightarrow$ *thesis*
    from $\langle fact \rangle$ **show** *thesis*
      apply (*insert that*) $\langle proof \rangle$
  qed
  fix $\vec{x}$ **assm** «*eliminate reduction*» q_1: $\vec{\varphi}_1$ **and** ... **and** q_n: $\vec{\varphi}_n$

To understand this definition of **obtain**, first observe that we have only refined the proof of the *reduction* statement. So the explanations given for the simplified version before are still valid due to compositionality of Isar proof checking.

The initial **proof** step really does nothing yet; any facts indicated for immediate use are absorbed here. The subsequent proof body proceeds in the canonical

fashion to establish a nested rule statement (cf. §5.2.5): we fix an arbitrary *thesis*, assume the premise *that* of the reduction (declared for implicit use as introduction rule), and claim the remaining goal. The proof of the latter captures the original facts and inserts the *that* part just before entering the remaining soundness proof as given in the original text. That additional tweak involving **apply** is intended to make automated proof tools behave more gracefully: the resulting goal of $(\bigwedge \vec{x}.\ \vec{\varphi}_1\ \ldots\ \vec{\varphi}_n \Longrightarrow \mathit{thesis}) \Longrightarrow \mathit{thesis}$ is essentially the same as the initial *reduction* statement, with all relevant information already present in the internal proof state. This is how common automated proof tools expect a situation to be solved with a single stroke (see also §7.3); if all fails, one may still refer to *that* explicitly in proof method specifications.

In contrast, single-step proof usually requires the individual constituent parts of the *reduction* made available as separate elements. Note that due to monotonicity of basic rule application (§2.4), the initial goal as modified by **apply** does not affect single *rule* steps at all.

In order to see soundness proofs of **obtain** in action, suppose we have certain standard elimination rules declared in the current context, especially $\exists$ and $\wedge$.

$$\exists x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow C) \Longrightarrow C$$
$$A \wedge B \Longrightarrow (A \Longrightarrow B \Longrightarrow C) \Longrightarrow C$$

Ideally, we would expect to satisfy the obligation posed by **obtain** after having performed a single elimination step modeled after the above statements. This almost happens to work out, although it requires an additional step inside.

assume $\exists x.\ P\ x$
then obtain x **where** $P\ x$
proof
  fix x **assume** $P\ x$
  thus *thesis* **by** (*rule that*)
qed

assume $A \wedge B$
then obtain A **and** B
proof
  assume A **and** B
  thus *thesis* **by** (*rule that*)
qed

Fortunately, assumption steps like "(*rule that*)" are already covered by Isar's builtin notion of solved goal configurations (§3.2.3). So we may actually collapse the above proofs to a single rule step "**..**" to achieve a succinct presentation.

assume $\exists x.\ P\ x$
then obtain x **where** $P\ x$ **..**

assume $A \wedge B$
then obtain A **and** B **..**

Such single step proofs work just the same for any statement χ that provides an elimination scheme $\vdash \chi \Longrightarrow (\bigwedge \vec{x}.\ \vec{\varphi} \Longrightarrow C) \Longrightarrow C$. As usual in Isar, there is nothing special about standard logical connectives, apart from being already declared in the standard theory library. For example, common schemes involving set-theory operators of Isabelle/HOL (see chapter 7) may look like this.

assume $x \in \bigcup C$
then obtain A **where** $x \in A$ **and** $A \in C$ **..**

assume $a \in \mathit{Domain}\ R$
then obtain b **where** $(a,\ b) \in R$ **..**

assume $b \in \mathit{Range}\ R$
then obtain a **where** $(a,\ b) \in R$ **..**

Note that the single rule schemes discussed so far did not yet employ the "[*intro*]" declaration of *that* given in the definition of **obtain** above. Introduction patterns employing this feature are less frequently encountered in practice, and shall be discussed later on (see §5.3.3).

5.3.3 Common patterns of generalized elimination

Canonical eliminations

We have already seen single step eliminations involving standard connectives. Once that multiple elements are encountered we better make use of existing proof tools for first-order logic to perform routine steps automatically.

assume $\exists\, x\ y\ z.\ P\ x \wedge Q\ y\ z \wedge R\ z$
then obtain $x\ y\ z$ **where** $P\ x$ **and** $Q\ y\ z$ **and** $R\ z$ **by** *blast*

Incidently, this version with explicit existential statements preceding **obtain** does not quite represent the most practical pattern yet. The above text basically contains two copies of the constituent propositions $P\ x$ and $Q\ y\ z$ and $R\ z$, which may be rather unwieldy expressions in reality. In realistic situations the existential claim typically emerges from a different fact in a more or less "trivial" fashion, mediated by standard automated tools (see also §7.3).

assume *something*
hence $\exists\, x\ y\ z.\ P\ x \wedge Q\ y\ z \wedge R\ z$ **by** *auto*
then obtain $x\ y\ z$ **where** $P\ x$ **and** $Q\ y\ z$ **and** $R\ z$ **by** *blast*

Furthermore, this pattern is usually better expressed without mentioning the intermediate existential statement in the first place.

assume *something*
then obtain $x\ y\ z$ **where** $P\ x$ **and** $Q\ y\ z$ **and** $R\ z$ **by** *auto*

This simplification assumes that the single automated proof step given here manages to bridge that gap directly. In general, the behavior of automated proof

tools may change significantly by inserting intermediate claims. Suppressing these as proposed above may make a big difference in the complexity of the new situation. On the other hand, the present situation is a very special case, where the intermediate existential statement is essentially just a different expression of the main obligation at hand; $\bigwedge C.\ (\bigwedge x\ y\ z.\ P\ x \Longrightarrow Q\ y\ z \Longrightarrow R\ z \Longrightarrow C) \Longrightarrow C$ and $\exists x\ y\ z.\ P\ x \wedge Q\ y\ z \wedge R\ z$ are even treated the same by typical automated proof tools like *blast* or *auto* (due to internal Skolemization).

Interestingly, the more compact form above actually works even better than expected in many situations. Existential quantifiers cause additional overhead for simple proof tools that may already be sufficient for the true content of the soundness proof. Avoiding $\exists$ in the first place, highly complex methods like *auto* may be replaced by plain rewriting of *simp*, for example (see also §7.3).

assume *something*
then obtain $x\ y\ z$ **where** $P\ x$ **and** $Q\ y\ z$ **and** $R\ z$ **by** *simp*

Speaking in terms of proof theory rather than automated reasoning, **obtain** acts pretty much like *cut elimination* of existential statements: instead of existential introductions followed by eliminations, we just proceed directly from the contributing facts to the eliminated form of obtained results. This reduces both the complexity of primitive proofs, which are not directly encountered in Isar anyway, and simplifies mechanized proof processing. It also enables the primary proof text to express that reasoning more succinctly.

So we have actually encountered a rare coincidence of substantial simplifications at different conceptual levels of formal proof at the same time. Commonly the issues of primitive inferences versus primary proofs (§1.4) need not be directly related, or may be even complementary to each other.

Introduction proofs

The **obtain** element may be really understood as a generalized form of $\exists$ and $\wedge$ in both directions. In particular, we may also perform introduction proofs, as illustrated by the following basic patterns.

obtain x **where** $P\ x$ ⟨*proof*⟩
hence $\exists x.\ P\ x$ **..**

obtain A **and** B ⟨*proof*⟩
hence $A \wedge B$ **..**

In typical applications of such schemes, the obtained parameter x stems from another (explicit or implicit) existential fact established earlier. Thus we accommodate notoriously difficult reasoning of the kind $\exists x.\ P\ x \Longrightarrow \exists y.\ Q\ y$, even without requiring either quantification to be stated explicitly (e.g. see the application of chapter 10, especially the main invariance proof in §10.6.3). Furthermore, the conjunctive form above achieves the effect of multiple simultaneous results (see also the related discussion in §9.4.1).

Apart from changing its external use, we may also reverse the internal standard procedure of soundness proofs of **obtain**, replacing the previous scheme of elimination–introduction by introduction–elimination.

obtain x **where** $P\ x$
proof
  show $P\ a$ ⟨*proof*⟩
qed

obtain A **and** B
proof
  show A ⟨*proof*⟩
  show B ⟨*proof*⟩
qed

Note that the initial **proof** steps encountered here actually use the "[*intro*]" declaration of *that* given in our full definition of **obtain** (§5.3.2). After backchaining of $\vdash \bigwedge x.\ P\ x \Longrightarrow \mathit{thesis}$ or $\vdash A \Longrightarrow B \Longrightarrow \mathit{thesis}$, we are required to exhibit an explicit existential witness or solve two conjuncts, respectively. This pattern may be generalized to several parameters and assumptions, without requiring advanced proof tools (the *that* rule ranges simultaneously over the new context).

obtain $x\ y\ z$ **where** $P\ x$ **and** $Q\ y\ z$ **and** $R\ z$
proof
  show $P\ a$ ⟨*proof*⟩
  show $Q\ b\ c$ ⟨*proof*⟩
  show $R\ c$ ⟨*proof*⟩
qed

Little more needs to be said about introduction of the conjunction case, it merely results in a different arrangement of the same reasoning performed before. In contrast, there is a fundamental difference with existential parameters getting involved: after the initial introduction we are left with a stronger problem to be solved in the present context. In particular, we cannot just produce an abstract existential fact and eliminate it for the witness of "**show** $P\ a$", as this would violate the scoping rules of parameters. The following failed attempt documents this common error of beginners when reasoning with existential statements.

obtain x **where** $P\ x$
proof
  have $\exists\, x.\ P\ x$ ⟨*proof*⟩
  thus $P\ a$
    ⋮

In order to finish that proof, one would essentially require a choice principle of the underlying object-logic, in order to extract a witness from an abstract existential statement "out of scope", so to say. Incidently, the version of HOL used in the Isabelle/HOL application environment (see chapter 7) does provide

Hilbert's choice operator (see §8.5), which could be used to fix our present mistake. On the other hand, we certainly would not like to assume strong choice principles for arbitrary object-logics of the Isabelle/Isar framework. This way of repairing broken elimination proofs via choice principles is certainly *not* intended as the primary use of the present introduction scheme of **obtain**.

A more useful application is to achieve a different proof layout in particular situations where the writer wishes to hide large proofs of existential introduction inside the body of the soundness proof. The main reasoning may then proceed with abstract existential parameters and their characteristic assumptions in a fully abstract manner. This is just another instance of the existential cut-elimination observed before, we may directly manipulate generalized existential statements in the proof text without ever needing explicit $\exists$ quantification.

Obtained parameters in forward proof

The **obtain** command has been defined as a derived context element of the basic Isar framework of natural deduction (cf. §3.3.1). In particular, **obtain** is not directly dependent on a goal configuration. Any number of results may be exported from its scope provided that its local existential parameters are not exposed to the outer context.

The Isabelle/Isar system implementation [Wenzel, 2001a] performs an explicit check of the proviso of existential parameters (cf. the inference rule *eliminate* given in §5.3.1). This achieves meaningful error messages of incremental proof processing. Otherwise the user would get a low-level failure of the underlying inference kernel of Isabelle/Pure that is quite hard to trace to the corresponding **obtain** language element located somewhere farther upwards in the proof text. It is important to note that this high-level checking of side-conditions is just a matter of user-convenience. Due to "manifest soundness" (§1.3) Isar proof processing may never produce "wrong" theorems in the first place.

The export behavior of **obtain** may be observed without being distracted by pending goals, using raw proof blocks again as a "logic laboratory" (cf. §5.2.4).

{
  obtain x **where** $P\ x$ ⟨*proof*⟩
  have C ⟨*proof*⟩
} — *this* $= \vdash C$

There is no need to export a single result C statement from the block, but only ensure that any fact produced there do not mention obtained parameters. In particular, the block may not be closed right after **obtain** itself, as this would be an illegal attempt to export $\vdash P\ x$ directly. There is no problem to export a conclusion like $\vdash \exists x.\ P\ x$, where the existential parameter is again being bound.

{
  obtain x **where** $P\ x$ ⟨*proof*⟩

hence $\exists x.\ P\ x$ **..**
} — *this* $= \vdash \exists x.\ P\ x$

Incidently, the fundamental inability to export obtained facts immediately is the deeper reason why there is nothing like **obtain-goal** in Isar, which would include solving of enclosing goals, just like **show** opposed to **have** (§3.2.3 and §5.2.2). This variant form of **obtain-goal** would never work with existential parameters, limiting its advantage over the existing **show** element to simultaneous conclusions. The latter feature is not much of an improvement either, since multiple sequential **show** statements already achieve a similar effect.

Apart from **obtain**, we have already covered simpler derived context elements of Isar (§3.3.1 and §5.2.1). This raises the question of how these are related.

First of all, **assume** and **presume** (which are the same up to the exact effect on an enclosing goal context) introduce immediate premises, which induces an additional hypothesis on any exported result (cf. §3.3.1). Assumptions like this are not "proven", but typically become admissible in particular situations, e.g. as the effect of an earlier backward step. In contrast, **obtain** involves an explicit proof that a number of facts (with parameters) may be assumed just now, such that self-contained results may get rid of the pending hypothesis.

The **def** declaration (cf. §3.3.1) refers to intro-logical definitional extensions of the proof context. The same idea may be represented via the general conservative extension mechanism of **obtain**, mediated via reflexivity and (implicit) substitution of "≡". So we may relate **obtain** and **def** as follows.

{
def $x \equiv t$
have $P\ x$ ⟨*proof*⟩
} — *this* $= \vdash P\ t$

{
obtain x **where** *x-def*: $x \equiv t$
proof
show $t \equiv t$ **by** (*rule reflexive*)
qed
have $P\ x$ ⟨*proof*⟩
hence $P\ t$ **by** (*unfold x-def*)
} — *this* $= \vdash P\ t$

Representation proofs

Representations of existing elements in terms of other concepts are a common theme in many applications. A typical example of representation in plain mathematics is illustrated by the following examples.

assume $y \in$ *range* f
then obtain x **where** $y = f\ x$ **..**

assume *surj f*
then obtain x **where** $a = f\ x$ **..**

Such representation patterns are actually just further instances of the general elimination form of **obtain** already seen before. Nevertheless, we may gain some further understanding of common patterns of informal reasoning with "elements of the form of something". In mathematical proofs one would usually refrain from stating any kind of explicit existential statement in between, but simply proceed with the obtained representation in a casual manner. Here **obtain** turns out as an adequate formal representation.

Slightly more concrete representations are frequently encountered in computer-science applications, typically involving concrete syntactic models (e.g. see chapter 10). The most basic instance essentially proceeds as follows.

fix $x\ y\ z$ **assume** $(a,\ b,\ c) = (x,\ f\ y,\ z)$ **and** $P\ (f\ y)$
then obtain d **where** $b = f\ d$ **and** $P\ b$ **by** *simp*

Here the **fix**/**assume** part is typically not stated explicitly in the text, but stems from a cases or induction scheme (see also §5.4 and §7.2.1). There are usually several new parameters like x, y, z that become redundant after some "obvious" simplifications. The above **obtain** pattern extracts the key representation properties succinctly, hiding superfluous parameters within the atomic proof step of "**by** *simp*".

Purely syntactic representations are occasionally encountered as well, usually related to inductive datatypes (see also §7.2.1). Consider these trivial examples.

assume $0 < n$
then obtain m **where** $n = Suc\ m$ ⟨*proof*⟩

assume $1 < n$
then obtain m **where** $n = Suc\ (Suc\ m)$ ⟨*proof*⟩

We see that **obtain** supports numerous useful proof patterns, providing a high-level view on general elimination rules $\vdash \vec{A} \Longrightarrow (\bigwedge \vec{x}.\ \vec{B}\ \vec{x} \Longrightarrow C) \Longrightarrow C$, with a single "case" of $\bigwedge \vec{x}.\ \vec{B}\ \vec{x} \Longrightarrow C$ and an identical conclusion C. Different proof techniques are required once that several cases or even recursive ones get involved. The corresponding concepts of case-analysis and induction shall be introduced in §5.4. These will be centered around the specific proof methods of *cases* and *induct*, which are may be used together with **case**.

5.4 Proof by cases and induction

5.4.1 Immediate patterns of cases and induction

Since Isar is based on a generic higher-order framework there is in principle nothing special about proof by cases and induction. First of all, an initial

problem may be split into several sub-problems just by using an appropriate rule, e.g. the standard one of $\vee$ elimination. As is typical for case rules the main thesis is preserved, but additional local assumptions emerge in each branch.

assume $A \vee B$
hence C
proof
  assume A
  thus *?thesis* ⟨*proof*⟩
next
  assume B
  thus *?thesis* ⟨*proof*⟩
qed

Note that there is nothing special about the **next** command (§3.2.3), it merely provides a succinct form to manage separate blocks within the proof body. Blocks are usually required in applications of case-split rules since each sub-problem may assume its own local context. This basic structure is made explicit below, where the text mimics the rule $\vdash A \vee B \Longrightarrow (A \Longrightarrow C) \Longrightarrow (B \Longrightarrow C) \Longrightarrow C$ more directly.

assume $A \vee B$
hence C
proof
  { assume A **thus** *?thesis* ⟨*proof*⟩ **}**
  { assume B **thus** *?thesis* ⟨*proof*⟩ **}**
qed

Cases need not depend on major premises to be eliminated, but may naturally arise from inherent properties of the underlying structure (of types) as well, like boolean case split of classical logic illustrated below.

have C
proof (*rule case-split*)
  assume A
  thus *?thesis* ⟨*proof*⟩
next
  assume $\neg\ A$
  thus *?thesis* ⟨*proof*⟩
qed

Case rules may introduce local (existential) parameters, too. Consider the following pattern involving the canonical non-recursive representation of the type of natural numbers (see also §7.2.1).

have C
proof (*rule nat.exhaust*)
  assume $n = 0$
  thus *?thesis* ⟨*proof*⟩

```
next
  fix m assume n = Suc m
    — existential parameter m only occurs in local assumption
  thus ?thesis ⟨proof⟩
qed
```

Induction rules are similar to the ones for plain cases encountered so far, but involve a few further issues. In particular, there are (universal) inductive parameters occurring in the conclusion. Thus the main thesis is not preserved as before, but is subject to the inductive structure of the logical entities involved. Induction proofs may in principle be performed via basic rule applications (§3.3.2), but it is generally a good idea to provide an explicit instantiation (§3.3.2) of the induction parameter (or the predicate), in order to avoid unexpected results of higher-order unification (§2.4).

```
have P n
proof (rule nat.induct [of P n])
  show P 0 ⟨proof⟩
next
  fix n assume P n
  thus P (Suc n) ⟨proof⟩
qed
```

Here induction over natural numbers has been presented as an introduction pattern since it directly refers to an inherent property of the underlying type structure. Inductions in elimination form (involving explicitly chained facts) typically occur for inductive sets (see §7.2.1). For example, consider the following pattern of reasoning over the set of finite sets. The **inductive** definition of *Finites* used below is from the main Isabelle/HOL library (see §7.4).

```
assume A ∈ Finites
hence P A
proof (rule Finites.induct)
  show P {} ⟨proof⟩
next
  fix a :: 'a and A :: 'a set
  assume A ∈ Finites and P A
  thus P (insert a A) ⟨proof⟩
qed
```

In this form, the syntactic instantiation of the induction rule has been replaced by an explicit membership assumption chained into the rule method.

As illustrated by the previous proof patterns, we see that case splits and induction schemes may be directly expressed within the existing Isar framework. The inherent capabilities of the Isar proof processor already cover handling of separate sub-problems, as well as local contexts arising in individual cases (§3.2.3). Furthermore, the higher-order nature of the underlying framework (§2.2) and

its basic operations of higher-order back-chaining of arbitrary rules (§2.4) accommodate general induction schemes nicely. So we could in principle conclude the exposition of proof by cases and induction at that stage.

On the other hand, several issues prevent the present techniques from scaling up to larger applications. This includes minor annoyances, such as slightly low-level method specifications of "(*rule nat.exhaust*)" or "(*rule nat.induct* [*of P n*])". More serious problems are those of large local contexts arising from inductive definitions in typical computer-science applications (e.g. chapter 10), and the inadequate treatment of non-atomic induction predicates experienced by naive use of higher-order backchaining involved in the *rule* method.

Subsequently we outline a few simple additions to the basic Isabelle/Isar setup considered so far, in order to support proof by cases and induction more conveniently. This merely requires a few additional proof methods and attributes.

5.4.2 Rules and cases

The basic *rule* method (§3.3.2) does not differentiate any particular format of meta-level theorems, but performs higher-order backchaining uniformly (§2.4), with automatic lifting over local goal contexts and higher-order unification. A slightly more specific view on rules will be required by the *cases* and *induct* methods to be introduced later on (see §5.4.3). In particular, these methods will produce declarations of named cases (using the general version of the interpretation function $\mathcal{M}$ in §3.2.3). Having invoked such a method initially, users may refer to local contexts conveniently via the **case** command (§3.3.1).

Taking previous cases and induction schemes as a model, the general rule format to be considered is $\vdash A \Longrightarrow \ldots \Longrightarrow (\bigwedge \vec{x}.\ \vec{B}\ \vec{x} \Longrightarrow D\ \vec{x}) \Longrightarrow \ldots \Longrightarrow C$. Here A marks a certain prefix of "major premises", its length is specified via the *consumes* attribute. In practice, we merely encounter "*consumes* 0" for rules associated with types, and *consumes* 1 for sets. The remainder consists of several sections of "cases" $\bigwedge \vec{x}.\ \vec{B}\ \vec{x} \Longrightarrow D\ \vec{x}$. Each case is associated with a name, as provided by the *case-names* attribute. In practice, the name coincides with that of datatype constructors or introductions of inductive sets, respectively (see also §7.2.1). The particular terminology of case parameters $\vec{x}$ may be exploited in some situations, notably "open" induction patterns given in §5.4.4, the *params* attribute attaches specific names, e.g. those of a corresponding **primrec** definition (see §7.2.2).

The main conclusion C and the local ones $D\ \vec{x}$ are not treated specifically so far, although this might be relevant to the actual proof methods applied. Recall that rules for (non-recursive) cases just have $D\ \vec{x} = C$ everywhere. In induction patterns, the individual conclusions of $D\ \vec{x}$ typically consist of the particular "constructor" schemes of the underlying structure, with universal inductive parameters actually occurring in $D\ \vec{x}$, and apart from some existential ones covered by the assumptions $\vec{B}\ \vec{x}$ only.

Rules that have been decorated by such additional structural information may be declared for automatic use with the corresponding methods. Isabelle/Isar provides separate *cases* and *induct* attributes, each one supports either type or set rules: "*cases type*: *c*", "*cases set*: *c*", "*induct type*: *c*", "*induct set*: *c*". See also [Wenzel, 2001a] for further details on any of these attributes.

Users rarely need to declare rules themselves, but may rely on the existing Isabelle/HOL environment to take care of this for the standard definitional concepts, like **inductive** and **datatype** (see §7.2.1), or **typedef** (see §7.1.2). Nevertheless, rules need to be redeclared occasionally, e.g. see the modified representations involved in quotient types and rational numbers in chapter 9.

5.4.3 Proof methods

The *cases* and *induct* methods provide a uniform interface to case analysis and induction over types and sets, based on appropriate rule declarations in the current context (§5.4.2). Specific support is provided for implicit selection of rules, separate instantiations, and symbolic case names for use with the **case** command (§3.3.1). These declarations accommodate succinct specification of standard proof patterns to be covered later on (see §5.4.4 and §7.2.1).

We refrain from detailed definitions of the *cases* and *induct* methods. Essentially they provide a heavily sugared view of the basic *rule* method (§3.3.2). Here we merely outline the basic format of method specifications, see [Wenzel, 2001a] for further details. The syntax of both methods includes arguments for several terms, providing an explicit instantiation of the rules involved, and guiding rule selection from types. Rules may be also given explicitly by the user.

The full method expression "(*cases* $\vec{t}$ *rule*: *r*)" refers to case-analysis of objects $\vec{t}$ via rule *r*. In partial specifications the exact reasoning pattern is determined as follows, depending on the type of arguments or chained facts.

facts		arguments	selected rule
	cases		classical case split
	cases	$t :: \tau$	standard cases of type τ
$\vdash t \in A$	*cases*	...	standard cases of set A
...	*cases*	... *rule*: *r*	cases by rule *r*

Previously declared rules (§5.4.2) may be also referred as "*type*: *c*" or "*set*: *c*".

A method expression "(*induct P* $\vec{x}$ *rule*: *r*)" is analogous to *cases*, but refers to induction over elements $\vec{x}$ using the (optional) predicate *P*. Special provisions are included to make induction work with non-atomic statements (see §5.4.5). In partial method specifications the induction rule is determined as follows.

facts		arguments	selected rule
	induct	$x :: \tau$	standard induction of type τ
$\vdash x \in A$	*induct*	...	standard induction of set A
...	*induct*	... *rule*: *r*	induction by rule *r*

Having selected (and instantiated) an appropriate rule as indicated above, these proof methods extract the collection of named local contexts (the "cases", cf. §5.4.2), which are then emitted into the enclosing proof context. In order to be usable via **case** later on, cases need to be fully instantiated by means of the original method specification. Unbound schematic variables (stemming from the original rule) may render individual cases invalid, e.g. due to a specification of "(*induct x*)" that lacks the predicate instantiation.

Local parameters of cases are marked as "hidden" by default, inhibiting **case** to make new parameters appear in the text out of nothing. This "pure" behavior of the *cases* and *induct* method may be disabled by including the "(*open*)" option. The practical impact on these details will be covered later on (see §5.4.4).

Facts presented to either method are consumed according to the number of "major premises" of the rule (cf. §5.4.2), usually 0 for types and 1 for sets. Any additional fact is inserted into the goal verbatim before applying the rule. This allows facts to be split across *cases*/*induct* and a suitable followup method, as in the common idiom "**by** *cases auto*" (see also chapter 10 for applications).

5.4.4 Common patterns of cases and induction

We now reconsider a few basic patterns of proof by cases and induction (cf. §5.4.1), this time using the general infrastructure provided by the specific proof methods introduced before. See chapter 7 for further concrete schemes emerging from the particular environment of Isabelle/HOL. Plenty of examples are provided by existing Isabelle/Isar applications, e.g. see chapter 9 and chapter 10.

Classical case-distinction

In the following proof pattern we perform classical case-distinction succinctly, without naming the *case-split* rule explicitly as before (§5.4.1).

have *C*
proof *cases*
  assume *A*
  thus *?thesis* ⟨*proof*⟩
next
  assume ¬ *A*
  thus *?thesis* ⟨*proof*⟩
qed

Apart from stating local assumptions in the text, we may use the infrastructure of symbolic cases provided by the specific proof method setup (cf. §5.4.3). This requires the proposition to be specified beforehand. Below the facts emerging from the cases *True* and *False* have been included in the text. Recall that any facts emerging from "**case** *a*" happen to be named after *a* as well (cf. §3.3.1).

have C
proof (*cases* A)
  case *True* — *True* $= \vdash A$
  thus *?thesis* ⟨*proof*⟩
next
  case *False* — *False* $= \vdash \neg\ A$
  thus *?thesis* ⟨*proof*⟩
qed

The latter version appears as slightly more economic for large propositions, since it requires only a single occurrence of A in the text. On the other hand, that symbolic form turns out as as less comfortable in common situations where A is actually quite simple anyway, e.g. an equation $x = y$. Here it is more lucid to spell out the individual cases explicitly as before.

have C
proof *cases*
  assume $x = y$
  thus *?thesis* ⟨*proof*⟩
next
  assume $x \neq y$
  thus *?thesis* ⟨*proof*⟩
qed

Other case distinctions require to name a suitable rule, such as the one for linear orders $\vdash (x < y \Longrightarrow C) \Longrightarrow (x = y \Longrightarrow C) \Longrightarrow (y < x \Longrightarrow C) \Longrightarrow C$.

have C
proof (*cases rule*: *linorder-cases*)
  assume $x < y$
  thus *?thesis* ⟨*proof*⟩
next
  assume $x = y$
  thus *?thesis* ⟨*proof*⟩
next
  assume $y < x$
  thus *?thesis* ⟨*proof*⟩
qed

Again, we may rephrase that proof using symbolic case names (stemming from the *linorder-cases* rule), together with a full instantiation given beforehand.

have C
proof (*cases* x y *rule*: *linorder-cases*)
  case *less* — *less* $= \vdash x < y$
  thus *?thesis* ⟨*proof*⟩
next
  case *equal* — *equal* $= \vdash x = y$
  thus *?thesis* ⟨*proof*⟩
next

```
  case greater  — greater = ⊢ y < x
  thus ?thesis ⟨proof⟩
qed
```

Structural cases

Structural case-analysis typically involves a canonical discrimination of elements of inductive sets or types, according to the introduction schemes or constructors given in the original definition (see §7.2.1 for Isabelle/HOL specifics). The subsequent patterns discriminate over the cases of natural numbers according to the inductive structure of 0 and *Suc*. The first version below merely uses "(*cases n*)" as a succinct specification of that mode of reasoning, being slightly more abstract than "(*rule nat.exhaust*)" encountered before (§5.4.1).

```
have C
proof (cases n)
  assume n = 0
  thus ?thesis ⟨proof⟩
next
  fix m assume n = Suc m
  thus ?thesis ⟨proof⟩
qed
```

Alternatively, we may invoke the symbolic case names associated with this rule. The names happen to be those of the datatype constructors.

```
have C
proof (cases n)
  case 0
  thus ?thesis ⟨proof⟩
next
  case Suc
  thus ?thesis ⟨proof⟩
qed
```

Recall the standard policy of *cases* is to hide any local parameters emerging in the individual cases (§5.4.3). The assumption introduced for the *Suc* is something like $n = Suc\ m^\star$, for some hidden parameter name $m^\star$. For syntactic reasons there is no way to refer to that term directly, although the corresponding fact $Suc = \vdash n = Suc\ m^\star$ is readily available, e.g. it may contribute to an abstract existential conclusion as follows.

```
  case Suc
  hence ∃ m. n = Suc m ..
```

By including the "(*open*)" option in *cases* the local parameters from the original rule become available in the text. This form assumes that the terminology of parameters has been declared beforehand in a sensible manner (using the *params* attribute, cf. §5.4.2). For the type of natural numbers, parameters of cases

are derived from type names of constructor arguments, due to the **datatype** package of Isabelle/HOL (see also §7.2.1).

have C
proof (*cases* (*open*) n)

⋮

case *Suc*
hence $n = Suc\ nat$ **.**
thus *?thesis* ⟨*proof*⟩
qed

Here the parameter *nat* might appear as slightly unpleasant, both from its actual name and the way it emerges implicitly from the "**case** *Suc*" command. Essentially, this is the same problem as with `open` of modules in ML (e.g. [Paulson, 1991]) where previous definitions may intrude the current context unexpectedly. On the other hand, the implicit contexts stemming from individual rules in Isar are usually much smaller than ML library structures. Furthermore, the situation is generally better for **inductive** set definitions, where parameter names are derived from the original definition given by the user (see §7.2.1); **primrec** works in a similar fashion (see §7.2.2), although the rule needs to be referenced explicitly in the method specification.

Nevertheless, the "(*open*)" option could be considered harmful in many applications of *cases*. Interestingly, it may be avoided altogether in an elegant fashion by using **case** with **obtain** (see below). The situation is more subtle for induction, with (universal) parameters occurring in conclusions as well.

Cases and generalized elimination

Case-analysis rules consist of several clauses of the form $\bigwedge \vec{x}.\ \vec{B}\ \vec{x} \Longrightarrow C$ where the local parameters $\vec{x}$ do *not* occur in the local conclusion C, which also happens to be the same as the ultimate result (cf. §5.4.2). This particular scoping of parameters amounts to an (eliminated) existential statement, similar to the ones encountered before in "generalized elimination" via **obtain** (cf. §5.3). In fact, **obtain** may get used together with instances of **case** emerging from plain case-analysis to some advantage. We illustrate this by another version of the structural case-analysis on natural numbers.

have C
proof (*cases* n)

⋮

case *Suc*
then obtain m **where** $n = Suc\ m$ **..**
thus *?thesis* ⟨*proof*⟩
qed

Here "**case** *Suc*" produces again $\vdash\ n = \mathit{Suc}\ m^\star$, with a hidden existential parameter $m^\star$ (§5.4.3). The result is used in the subsequent soundness proof of **obtain**, according to the standard existential introduction pattern (cf. §5.3.3). As a consequence, we gain control over the terminology of parameters in the static proof text, rather than implicitly via **case** used with "(*cases* (*open*) ...)".

At first sight, we seem to have achieved very little compared to the more direct pattern of "**fix** m **assume** $n = \mathit{Suc}\ m$" seen before. Indeed the present technique of **case**/**obtain** really pays off in applications involving large case-analysis rules, typically those stemming from inductive sets with many side-conditions, or considerable structural overhead of the elements involved (tuples etc.). In such situations the original formulation of the local context is actually only of marginal interest. So it is better avoided in the proof text altogether in order to avoid excessive details distracting the reader.

The interesting properties emerging from a particular case are typically immediate consequences, after suitable simplification of superficial structure. This typically eliminates most parameters and assumptions in the first place, achieving important gain of overall readability. See chapter 10 for realistic applications of this technique; below we merely hint at such patterns in an abstract manner, essentially recounting a generalized elimination scheme already seen in §5.3.3.

case c
  — $\approx$ "**fix** $x^\star\ y^\star\ z^\star$ **assume** $(a,\ b,\ c) = (x^\star,\ f\ y^\star,\ z^\star)$ **and** $P\ (f\ y^\star)$ **and** ..."
then obtain d **where** $b = f\ d$ **and** $P\ b$ **by** *simp*

Here the soundness proof of **obtain** is again of the introduction form (§5.3.3), with the proof method *simp* taking care of obvious syntactic simplifications of tuple structures. As already pointed out earlier, this is an instance where the eliminated notion of existential statements of **obtain** considerably reduces the need of powerful reasoning tools. Explicit $\exists$ quantifiers in the statement would usually demand more sophisticated proof tools for first-order logic, like Isabelle's *blast* or *auto* instead of plain *simp* (see also §7.3).

Also note that the present **case**/**obtain** pattern may serve as a viable replacement for many incidents of Isabelle tactic scripts involving the `mk_cases` feature [Nipkow *et al.*, 2001] (the same is available in the tactic emulation of Isabelle/Isar via the **inductive-cases** command and the *ind-cases* method [Wenzel, 2001a]). The above Isar pattern is able to supplant such special ML tools, merely using the existing proof elements of **case**, **obtain**, and "**by** *simp*".

Structural induction

Common structural induction over datatypes is presented as an introduction pattern of the *induct* method, merely requiring a syntactic instantiation. For the particular datatype of natural numbers this coincides with common forms of "mathematical induction".

have $P\ n$

```
proof (induct n)
  show P 0 ⟨proof⟩
next
  fix n assume P n
  thus P (Suc n) ⟨proof⟩
qed
```

Alternatively we may refer to symbolic cases by giving a full instantiation and the "(*open*)" option. Note that in Isabelle/HOL the standard terminology of the parameter in mathematical induction is n, rather than the default one of *nat* imposed by the datatype package (see also §7.2.1).

```
lemma P (n::nat)
proof (induct (open) P n)
  case 0
  thus P 0 ⟨proof⟩
next
  case Suc
  thus P (Suc n) ⟨proof⟩
qed
```

In informal mathematics one encounters two different styles of introducing the local assumption in the induction step. The *active* style goes like "now assume that $P\ n$ holds (for an arbitrary but fixed n)", which closely corresponds to our previous formulation of "**fix** n **assume** $P\ n$". In contrast, the *passive* style is something like "we already have $P\ n$ (due to the induction hypothesis)". The latter may be expressed nicely in the **case** version of induction as follows.

```
  case Suc
  have P n by assumption  — "passive" assumption
  thus P (Suc n) ⟨proof⟩
```

Recall that proper assumptions must be introduced before any corresponding **show** (§5.2.1). This restriction does not appear to hold for passive ones since the context has already been augmented earlier by a previous **case** element.

```
  case Suc
  show P (Suc n)
  proof −
    have P n by assumption  — late reference of "passive" assumption
    thus ?thesis ⟨proof⟩
  qed
```

Here we have used "**by** *assumption*" for clarity; plain "**.**" works as well since the methods *assumption* and *this* coincide in pure introduction usage (§3.3.2).

Rule induction

By "rule induction" we refer to the canonical induction schemes emerging from inductive sets (see also §7.2.1). The introduction rules given there act very

much like constructors of an inductive type, although additional conditions may be included in set constructions. Furthermore, there are explicit membership judgments involved as separate facts, so induction appears as an elimination pattern. Subsequently, we recast the immediate pattern of rule induction of §5.4.1 (involving the set of finite sets) by using the generic *induct* method.

assume $A \in \mathit{Finites}$
hence $P\ A$
proof *induct*
  show $P\ \{\}$ ⟨*proof*⟩
next
  fix $a :: 'a$ **and** $A :: 'a\ \mathit{set}$
  assume $A \in \mathit{Finites}$ **and** $P\ A$
  thus $P\ (\mathit{insert}\ a\ A)$ ⟨*proof*⟩
qed

Here we have merely replaced the original "(*rule Finites.induct*)" method by plain *induct*. In the next version we actually use the symbolic cases provided by that method as well. Again we need to give a full instantiation beforehand (cf. §5.4.3) and declare local parameters as "(*open*)".

assume $A \in \mathit{Finites}$
hence $P\ A$
proof (*induct* (*open*) $P\ A$)
  case *emptyI*
  thus $P\ \{\}$ ⟨*proof*⟩
next
  case *insertI*
  thus $P\ (\mathit{insert}\ a\ A)$ ⟨*proof*⟩
qed

Avoiding unexpected parameters

The "(*open*)" form of induction patterns involving symbolic cases is often inappropriate, local parameters need to be specified more explicitly in the text to prevent confusion of readers. In principle, this could be achieved via more elaborate versions of the initial method invocation, using the *params* attribute (§5.4.2) as in "(*induct* (*open*) ... *rule*: *r* [*params x* ...])". This turns out as slightly too impractical for direct use, although the **primrec** package of Isabelle/HOL already provides renamed rules for immediate use (see also §7.2.2).

Recall that the situation is simpler for non-recursive *cases* where parameters may only occur existentially. Using **case** together with **obtain** as demonstrated before, hidden existential parameters may be easily extracted in the text.

Subsequently, we propose a different approach to explicit parameter specification that appears to be suitable for inductive situations as well. Concerning the *induct* method itself parameters are again hidden, but may be named on invocation of a slightly enhanced version of **case**. The following pattern illustrates

this idea (this proof does not yet work with the present version of Isabelle/Isar).

have $P\ n$
proof (*induct* $P\ n$)
  case 0
  thus $P\ 0$ ⟨*proof*⟩
next
  case (*Suc* n) — parameters included in case specification
  thus $P\ (Suc\ n)$ ⟨*proof*⟩
qed

An extended **case** specification of the form "$(c\ \vec{x})$" shall refer to a particular terminology to be used in that instance of case c in the text.

The above pattern appears to be quite promising, but still needs to be evaluated in concrete examples, including large inductive definitions (see §7.2.1).

5.4.5 Induction with non-atomic statements

The issue of induction with non-atomic rule statements is very important for non-trivial applications, due to additional conditions and variable parameters that are typically required in "strengthened" inductive statements (e.g. see the exposition in [Nipkow and Paulson, 2001]). Here we consider the standard mathematical induction rule $P\ 0 \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ n)) \Longrightarrow P\ n$, which gives rise to the following proof pattern (§5.4.4).

have $P\ n$
proof (*induct* n)
  show $P\ 0$ ⟨*proof*⟩
next
  fix n
  assume $P\ n$
  show $P\ (Suc\ n)$ ⟨*proof*⟩
qed

On the other hand, non-atomic rule statements may be established in Isar as follows, essentially just by replacing $\bigwedge$ and $\Longrightarrow$ of the proposition by **fix** and **assume**/**show** in the proof text (cf. §5.2.5).

have $\bigwedge x.\ A\ x \Longrightarrow C\ x$
proof −
  fix x
  assume $A\ x$
  show $C\ x$ ⟨*proof*⟩
qed

Both of these techniques may be combined as follows, in order to establish a non-atomic statement by induction.

```
have ⋀x. A x n ⟹ C x n
proof (induct n)
  fix x
  assume A x 0
  show C x 0 ⟨proof⟩
next
  fix x and n
  assume ⋀x. A x n ⟹ C x n — induction hypothesis as a rule
  assume A x (Suc n) — assumption of the concluded rule
  show C x (Suc n) ⟨proof⟩ — conclusion of the concluded rule
qed
```

Here the two seemingly orthogonal concepts of induction and non-atomic propositions appear to be fully compositional. In retrospective, nothing special seems to be happening, the Isar proof just turns out as one might have expected in the first place. On the other hand, this rather painless approach to induction with non-atomic statements is actually a result of careful treatment of these issues within the environment of structured proof texts.

Existing tactical provers, most notably Isabelle [Paulson and Nipkow, 1994] and traditional HOL systems [Gordon and Melham, 1993], need to impose quite a few further technical details on users (as illustrated below). The Isabelle/HOL tutorial [Nipkow and Paulson, 2001] includes lengthy instructions for prospective users to get "strengthened" induction statements accepted by the system. In practice, such superficial formal overhead causes considerable distractions from the main problem of figuring out suitable generalizations of inductive arguments.

Subsequently, we illustrate the problems encountered in tactical induction proofs by means of Isar proof texts. First of all, we observe that the induction scheme may not just be applied naively to the original rule statement, since higher-order backchaining involved here is monotonic wrt. the goal context (§2.4).

```
have ⋀x. A x n ⟹ C x n
proof (rule nat.induct) — monotonic rule application
  fix x assume A x n — unchanged rule context
  {
    show C x 0 ⟨proof⟩
  next
    fix m assume C x m — limited induction hypothesis
    show C x (Suc m) ⟨proof⟩
  }
qed
```

Here the normal behavior of *rule* application got in our way, as we have actually intended to get access to the original context of $\bigwedge x.\ A\ x\ n \Longrightarrow \ldots$ within the induction, rather than as static assumptions outside (even over different parameters). The established technique to fix this misconception in unstructured tactic scripts is to rephrase the original rule as an atomic proposition via

object-level $\forall$ /$\longrightarrow$ connectives.

have $\forall x.\ A\ x\ n \longrightarrow C\ x\ n$
proof (*rule nat.induct*)
  show $\forall x.\ A\ x\ 0 \longrightarrow C\ x\ 0$ ⟨*proof*⟩
next
  fix n
  assume $\forall x.\ A\ x\ n \longrightarrow C\ x\ n$
  show $\forall x.\ A\ x\ (Suc\ n) \longrightarrow C\ x\ (Suc\ n)$ ⟨*proof*⟩
qed

While this looks fine in theory, it usually causes a considerable amount of additional formal noise in practice, for the following reasons.

1. $\forall$ /$\longrightarrow$ connectives need to be stripped by explicit introductions before entering the actual proof of an inductive conclusions.
2. $\forall$ /$\longrightarrow$ connectives need to be eliminated from the induction hypothesis to become usable in common situations.
3. The final result needs to be modified to recover the originally intended rule statement.

There is even a fourth (extra-logical) problem for the writer during development. Typically, the particular context to be passed through an induction is subject to some experimentation, until a properly strengthened statement has been figured out. Switching back and forth between $\bigwedge$/$\Longrightarrow$ and $\forall$ /$\longrightarrow$ happens to be quite cumbersome in Isabelle due to different syntactic precedences, so proper placement of parentheses is imposed on the writer as well.

Below we illustrate the first three (formal) issues by the following proof pattern. Recall that structural decomposition shown statically in the Isar text below would be hidden in extraneous tactic invocations in an unstructured proof script.

have $\forall x.\ A\ x\ n \longrightarrow C\ x\ n$
proof (*rule nat.induct*)
  show $\forall x.\ A\ x\ 0 \longrightarrow C\ x\ 0$
  proof — (1)
    fix x
    show $A\ x\ 0 \longrightarrow C\ x\ 0$
    proof — (1)
      assume $A\ x\ 0$
      show $C\ x\ 0$ ⟨*proof*⟩
    qed
  qed
next
  fix n
  assume $\forall x.\ A\ x\ n \longrightarrow C\ x\ n$

```
  hence ⋀x. A x n ⟶ C x n .. — (2)
  hence ⋀x. A x n ⟹ C x n .. — (2)
  show ∀x. A x (Suc n) ⟶ C x (Suc n)
  proof — (1)
    fix x
    show A x (Suc n) ⟶ C x (Suc n)
    proof — (1)
      assume A x (Suc n)
      show C x (Suc n) ⟨proof⟩
    qed
  qed
qed
note mp [OF spec [OF this]] — (3)
  — this = ⊢ ⋀x. A x n ⟹ C x n
```

The special attribute *rule-format* of Isabelle/Isar's tactic emulation [Wenzel, 2001a] simplifies phase (3) at the least. Concerning introductions (1) and eliminations (2), unstructured tactic scripts would not be blown up that excessively like the above text, but the same inherent complexity is still there. Users of tactical proving have suffered the additional overhead of simulating induction over rule statements for several years, but we certainly would not like to have that way persist in proper Isar. As illustrated here, any superficial formal noise immediately affects Isar proof texts in an unacceptable manner.

Fortunately, the whole affair is quite easy to accommodate in proper Isar, once that the compositional version of induction with non-atomic statements demonstrated initially has been recognized as the right way. The key observation is that *induct* merely needs to provide a filter for plain *rule*, in order to internalize a rule statement temporarily. This is how *induct* actually works in Isar, ignoring the separate issues of rule selection, instantiation, and providing named cases already covered in §5.4.3.

```
induct = (unfold atomize, rule r, fold atomize)
```

Here *atomize* contains $\vdash (\bigwedge x.\ P\ x) \equiv \forall^\star x.\ P\ x$ and $\vdash (A \Longrightarrow B) \equiv A \longrightarrow^\star B$, transforming any meta-level $\bigwedge/\Longrightarrow$ connectives to hidden reflections $\forall^\star/\longrightarrow^\star$ within the object-logic (private copies of $\forall/\longrightarrow$). As a consequence, the actual application of the induction rule r appears to operate on an atomic proposition. Afterwards the $\forall^\star/\longrightarrow^\star$ connectives are again expanded, in order to recover the presentation as a rule.

The latter technique assumes that occurrences of the induction predicate are in proper "rule positions", merely surrounded by $\bigwedge/\Longrightarrow$. The rules of existing Isabelle/HOL packages, most notably **datatype** and **inductive** (see §7.2.1), already follow this form. Only a few individual rules of the Isabelle/HOL library (see also §7.4) need to be adapted to become usable with rule statements, e.g. $(\bigwedge n.\ \forall m.\ m < n \longrightarrow P\ m \Longrightarrow P\ n) \Longrightarrow P\ n$ which has to be rephrased as $(\bigwedge n.\ (\bigwedge m.\ m < n \Longrightarrow P\ m) \Longrightarrow P\ n) \Longrightarrow P\ n$.

The above definition of the *induct* method appears as a rather obvious solution to a long standing inconvenience of classic Isabelle. Two key observations make our scheme practically useful in Isar proof texts, as opposed to unstructured tactic scripts. Firstly, Isar is able to treat non-atomic assumptions and goals in a uniform manner (§5.2.5), while Isabelle tactics have a strong bias towards flattened proof problems. Secondly, rule statements may be easily formulated as separate local facts in Isar, while tactic scripts operate on a single large "rule" that encodes the full proof problem with assumptions and intermediate facts, so the range of the actual inductive statement would not be delimited clearly.

We see that the additional effort of structured proof composition eventually pays off by significant simplification of typically subtle inductions with generalized statements, both from the conceptual and technical point of view. In particular, the common way to represent natural deduction proof schemes at the meta-level via $\bigwedge/\Longrightarrow$ carries over to "advanced" inductive problems as well. The former attempt to encode rules temporarily within the object-logic essentially gives up the fundamental conveniences of the generic Isabelle framework in higher-order backchaining of meta-level expressions (§2.4). That slightly strange exceptional case falling back to the object-logic for induction is apt to diminish the elegance of the original Isabelle framework [Paulson, 1989] [Paulson, 1990] unnecessarily.

5.5 Discussion

5.5.1 Context manipulations in Mizar

Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] provides a number of proof outline commands for procedural transformations of contexts (and goals). We have already covered the propositional case in §4.2.4, the issue of adequate treatment of quantifiers is even more important. Here we consider the generally critical scheme of $\exists x.\ P\ x \Longrightarrow \exists y.\ Q\ y$, exploring the particular instance of $\exists x.\ \forall y.\ R\ x\ y \Longrightarrow \forall v.\ \exists u.\ R\ u\ v$. The following Isar proof documents a formulation in basic natural deduction (cf. chapter 4), without any of those "advanced" techniques considered previously.

lemma $\exists x.\ \forall y.\ R\ x\ y \Longrightarrow \forall v.\ \exists u.\ R\ u\ v$
proof
  fix v
  assume $\exists x.\ \forall y.\ R\ x\ y$
  thus $\exists u.\ R\ u\ v$
  proof
    fix u
    assume $\forall y.\ R\ u\ y$
    hence $R\ u\ v$ **..**
    thus *?thesis* **..**
  qed
qed

In Isar the structure of basic natural-deduction proofs directly corresponds to that of the logical statements involved. This results in the typical nesting of sub-proofs encountered above.

Mizar provides separate commands to dig into complex statements in a sequential manner, notably `let` for $\forall$ introduction, `take` for $\exists$ introduction, `consider` for $\exists$ elimination from an intermediate result, and `given` for $\exists$ elimination from an existential premise. As in the propositional case (cf. §4.2.4), we follow the simplified view of Mizar operations here according to [Wiedijk, 2000]. Authentic Mizar treats quantifiers by hardwired classical proof principles. There is no official documentation available how this works exactly. Critical users cannot even find out for themselves, since the sources of Mizar are unavailable.

For the present example, we use the stripped-down version Mizer-MSE [Hoover and Rudnicki, 1996] [Prazmowski and Rudnicki, 1993] [Mizar MSE]. Unlike full Mizar, the MSE version supports abstract notation of first-order logic, which simplifies our presentation. On the other hand, it lacks the `given` variant of $\exists$ elimination, so we need to use plain `consider` here.

```
environ
  reserve x, y, u, v for a;
begin
  (ex x st for y holds P[x, y]) implies (for v holds ex u st P[u, v])
proof
  assume A: ex x st for y holds P[x, y];
  let v be a;
  consider u being a such that B: for y holds P[u, y] by A;
  C: P[u, v] by B;
  thus ex u st P[u, v] by C;
end;
```

Here the assumption `A` gets used later on to obtain the local parameter `u` via $\exists$ elimination of `consider`. Mizar needs to keep a fixed order of transformations when digging into a nested statement. Consequently, the intermediate `let` element may not be moved out of the way in order to link the `assume` and `consider` lines more directly. The fact `B` needs to be named explicitly as well, since Mizar treats the result of `consider` as a special case, disallowing immediate linking via `then`, for example. In fact, the Mizar-MSE version does not support Mizar's linked elements `then`/`hence`/`hereby` at all. For the very same reason, we also require the `C` label above in Mizar-MSE, where full Mizar could have employed `hence` instead of `thus` to use the previous fact directly.

We observe that Mizar tends to introduce some amount of formal noise due to slightly inflexible arrangement of its basic proof outline elements. In contrast, Isar proof contexts are invariant wrt. a number of canonical algebraic laws, like permuted assumptions, or commuted parameters and assumptions according to HHF normal forms (cf. §2.4.1 and §5.2.1).

On the other hand, Isar's non-procedural approach of proof contexts may require excessive nesting of sub-proofs in some situations. In practice, additional nesting

is often due to generalized elimination patterns, which may be accommodated more adequately by the derived **obtain** element (§5.3). Thus we may rephrase the present example in Isar without any nesting, but are still able to exploit the flexible treatment of context elements to arrange the flow of facts more naturally. This happens to keep the text free from explicit references to facts and rules.

lemma $\exists x.\ \forall y.\ R\ x\ y \Longrightarrow \forall v.\ \exists u.\ R\ u\ v$
proof
  fix v
  assume $\exists x.\ \forall y.\ R\ x\ y$
  then obtain u **where** $\forall y.\ R\ u\ y$ **..**
  hence $R\ u\ v$ **..**
  thus $\exists u.\ R\ u\ v$ **..**
qed

Even though Isar is quite far removed from direct manipulations of proof contexts (or goals), the particular technique of calculational reasoning provides a slightly different paradigm of implicit transformations of results, albeit in a very disciplined manner (see chapter 6).

5.5.2 Second-order schemes in Mizar and DECLARE

Both Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] and DECLARE [Syme, 1997a] [Syme, 1998] [Syme, 1999] treat second-order proof schemes like induction as a special case, although for different reasons.

Mizar is based on classical first-order logic, together with a hardwired axiomatization of typed set-theory. There is a special sub-language to handle second-order rules, including the top-level goal statement of `scheme`, and the `from` element for application of rule schemes in proofs. For example, the principle of mathematical induction is stated in article #4 (`NAT_1`) of [Mizar library] as follows.

```
scheme Ind  P[Nat]  :
 for k holds P[k]
   provided
A1:   P[0] and
A2:   for k st P[k] holds P[k + 1]
   proof ... end;
```

Having decomposed the main statement as indicated above, the proof proceeds by using standard proof elements of Mizar. The body will refer to plain first-order statements only, since rule schemes may not be nested deeper. Subsequently we give an example of scheme application in Mizar, taken from article #676 (`DYNKIN`) of [Mizar library]. The `from` language element is used here with the same rule `Ind` of mathematical induction.

```
theorem fin:
 for n holds PSeg(n) is finite
proof
 A0: PSeg(0) is finite by ALGSEQ_1:11;
 A1: for n st PSeg(n) is finite holds PSeg(n+1) is finite
 proof
  let n;
  assume PSeg(n) is finite; then
  PSeg(n) U {n} is finite by FINSET_1:14;
  hence thesis by ALGSEQ_1:17;
 end;
 thus thesis from Ind(A0,A1);
end;
```

This example represents the common idiom of induction in Mizar. Scheme application only works with existing facts provided as named arguments, as encountered in "`from Ind(A0,A1)`" above. So induction is restricted to forward reasoning, with explicit references to previous facts. Incidently, this extreme style of forward induction is also the preferred one of the (informal) discipline of writing long and detailed proof outlines proposed by [Lamport, 1994].

Isar is more flexible in several respects. First of all, the underlying framework supports complex rules directly (§2.2), without any artificial restriction to first-order logic. The Isar proof language accommodates this adequately, by treating non-atomic statements uniformly in the text (§5.2.5 and §5.4.5).

Furthermore, the freedom to choose the direction of reasoning freely may in principle be applied to "second-order" patterns as well, although our preferred style of induction is in backwards manner. Isar admits the following variations of mathematical induction, ranging from an emulation of Mizar, over a slightly odd mixed form, to our standard one of backward reasoning (cf. §5.4).

have 0: $P\ 0$ ⟨*proof*⟩
have *Suc*: $\bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ n)$
proof −
  fix n
  assume $P\ n$
  thus $P\ (Suc\ n)$ ⟨*proof*⟩
qed
from 0 **and** *Suc* **have** $P\ n$ **by** (*rule nat.induct*)

have $P\ 0$ ⟨*proof*⟩
hence $P\ n$
proof (*rule nat.induct*)
  fix n
  assume $P\ n$
  thus $P\ (Suc\ n)$ ⟨*proof*⟩
qed

have $P\ n$
proof (*induct* n)
  show $P\ 0$ ⟨*proof*⟩
next
  fix n **assume** $P\ n$
  thus $P\ (Suc\ n)$ ⟨*proof*⟩
qed

The backward version appears to be more conforming to the main-stream style of induction in mathematical proofs (cf. §5.4.4). Its "analytical" presentation in top-down fashion is especially well-suited for large-scale applications that demand additional infrastructure like symbolic cases (§5.4.2), or proper treatment of non-atomic inductive predicates (§5.4.5). The specific Isar infrastructure given in §5.4 requires the induction pattern to be specified in advance, in order to enable succinct presentation of the corresponding sub-proofs.

DECLARE [Syme, 1997a] [Syme, 1998] [Syme, 1999] follows a similar idea of top-down induction. The system provides a separate sub-language for rule specifications, by the builtin `proceed` element for "second-order schema application for inductive and other arguments". The following example is from the Java$_S$ case-study performed in DECLARE [Syme, 1998, part II].

```
thm array-alloc-conforms-lemma
if TE wf_tyenv  <TE_wf>
   TE |- VT(st,ext) wf_vartype  <st_wf>
   (val1,heap1) = val_alloc(heap0,st,dims,ext)  <alloc>
   TE |- heap0 heap_conforms  <heap0_conforms>

then heap0 FPSUBFUN heap1 &
      TE |- heap1 heap_conforms &
      (TE,heap1) |- val1 wconforms_to VT(st,LEN(dims)+ext)  <goal>;

proceed by structural induction on dims with dims,heap0,heap1,val1 variable;

case NIL dims = [];
  qed by ...;

case CONS
  dims = dim × dims'  <dims>;
  ...
end;
```

Here the rule has been specified as "`structural induction`" over lists (which is the type of `dims`), while the annotation "`with ... variable`" refers the selection of universal parameters in the induction. DECLARE performs implicit quantification of the inductive statement as specified. The "scope" of the resulting induction predicate is determined automatically, covering exactly those parts of the current proof context that mention any the parameters. This discipline enables DECLARE to reason about top-level sequents inductively.

DECLARE implements specific support for common induction patterns as part

of the primary language. This conforms to its overall approach of a specialized system for reasoning about operational semantics [Syme, 1998]. In Isar the aims and general philosophy are quite different, though. Great care has been taken in order to provide a generic framework for high-level natural-deduction proofs, with only minimal instantiations required to support many different kinds of applications. Any such advanced concepts are kept as close to pure logical concepts as sensible.

Concerning the particular case of induction proofs, this means that in Isar local parameters and assumptions are just given via meta-level connectives $\bigwedge/\Longrightarrow$ in the original claim, instead of separate language elements in DECLARE. The *induct* method of Isar passes exactly the given rule statement through the induction (§5.4.5), without any further implicit operations involved.

5.5.3 Generalized case-splitting

Two recurrent patterns of advanced natural deduction have been treated specifically so far: generalized elimination via **obtain** (§5.3), and an infrastructure for rules involving case-splitting (§5.4). Furthermore, we have already pointed out common use of **obtain** together with **case** (§5.4.4). The question remains if case-splitting may be incorporated into **obtain** directly as well.

Speaking in terms of the underlying framework of minimal higher-order logic (§2.2), **obtain** corresponds to singleton case-analysis (with optional existential parameters) expressed by the *reduction* statement $\bigwedge C.\ (\bigwedge \vec{x}.\ \vec{\varphi}_1\ \ldots\ \vec{\varphi}_n \Longrightarrow C) \Longrightarrow C$ (cf. §5.3.1 and §5.3.2). From the meta-logical perspective, the idea underlying **obtain** could be easily generalized to any number of branches. This would essentially result in *generalized case-splitting*, which we shall associate with the hypothetical **obtain-cases** element as follows.

obtain-cases
 a: $\vec{x}$ **where** $A_1\ \vec{x}$ **and** $A_2\ \vec{x}\ \ldots$ |
 b: $\vec{y}$ **where** $B_1\ \vec{y}$ **and** $B_2\ \vec{y}\ \ldots$ | $\ldots$
  ⟨*proof*⟩
 case a
 hence C ⟨*proof*⟩
next
 case b
 hence C ⟨*proof*⟩
next
 ⋮

Here the initial soundness proof would establish the multi-branch *reduction* statement of this particular case-split with separate existential parameters. The subsequent portions of the text may invoke the individual alternatives via **case**, and need to establish a common result C eventually.

Unfortunately, the existing Isar proof text processing scheme (§3.2.3) does not

support this kind of reasoning pattern as a context element like **obtain** (§5.3). The Isar/VM interpreter is essentially bound to a linear operation, working incrementally from left to right. For **obtain-cases**, the discharge operation of the underlying **assm** primitive (§3.2.1) would need to collect the results of several independent pieces of text, before being able to apply the covering statement proven beforehand.

Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] provides the language element "`per cases`" for this kind of isolated multi-branch reasoning, although it lacks local existential parameters. In fact, this is the only way that case-splitting may be performed in Mizar in the first place. The other logical manipulations covered so far (§4.2.4 and §5.5.2) operate in a strictly linear fashion. Recall that Mizar may not just apply arbitrary natural deduction rules like Isar's basic "**proof** (*rule r*)" form.

The following example of case-splitting in Mizar has been taken from article #638 (`polynom2`) of [Mizar library]. Incidently, this fragment uses purely forward reasoning at the outer level (via "`now ... end`"). The ultimate result is `contradiction`, but this need not be stated as a goal beforehand.

```
now per cases;
case N() = 0;
  hence contradiction by A4;
case N() <> 0;
  then 1 <= N() by RLVECT_1:99;
  then A6: Seg 1 c= Seg N() by FINSEQ_1:7;
  1 ∈ (Seg 1) by FINSEQ_1:4,TARSKI:def 1;
  hence contradiction by A5,A6;
end;
hence contradiction;
end;
```

The soundness proof required by "`per cases`" merely needs to establish a disjunctive statement (due to the lack of existential parameters in Mizar's case-splitting primitive). The empty soundness proof above (cf. the semicolon after "`per cases`") involves the classical principle of *tertium-non-datur*, which is already treated as "obvious" by Mizar's builtin verifier.

Concerning Isar, we have already observed that forward-style case-analysis like this is not directly supported by the existing language framework. Nevertheless, we may just use the most basic instance of the *cases* method applied to an initial claim, while still achieving a reasonable presentation (cf. §5.4). Thus the above Mizar text may be rephrased in Isar as follows.

have *False*
proof *cases*
  assume $n = 0$
  thus *False* ⟨*proof*⟩

next
assume $n \neq 0$
⋮
thus *False* ⟨*proof*⟩
qed

Interestingly, most practical applications of "logical" case-analysis in Mizar and Isar are actually of this simple *tertium-non-datur* form. In Isar there is another good portion of "structural" case-analysis of inductive sets and types (see also §7.2.1) that typically emerge in computer-science applications (e.g. chapter 10). Mizar does not offer any specific support in the latter respect. In fact, there are very few such examples in [Mizar library], which turn out as slightly low-level as users have to simulate the underlying inductive structures by raw elements of first-order logic and set-theory.

DECLARE [Syme, 1997a] [Syme, 1998] [Syme, 1999] provides a very general `cases` element, which refers to the "decomposition and enrichment" primitive of the system. This mechanism is quite powerful, admitting to split the proof context into several disjunctive parts, each introducing a number of local assumptions over (optional) existential parameters. Thus complex statements of the form $(\exists \vec{x}.\ A_1\ \vec{x} \wedge A_2\ \vec{x} \wedge \ldots) \vee (\exists \vec{y}.\ B_1\ \vec{y} \wedge B_2\ \vec{y} \wedge \ldots) \vee \ldots$ may be directly accommodated in DECLARE proofs. The system collects such covering statements from the individual portions of text given by the user as "`consider` $\vec{x}$ `such that` $A_1\ \vec{x}$ $A_2\ \vec{x}$ ..." elements (separated by `case` keywords). In contrast to Isar, DECLARE may achieve this relatively easily due its non-interactive processing of proof texts.

All other context elements of DECLARE are actually derived from the `cases` primitive, including local facts (`have`), single-ended elimination (`consider` without surrounding `cases`/`case`), and local definitions (`let`) with optional pattern matching performed *within* the logic. For example, local definitions work out analogously to the Isar pattern for **def** given for **obtain** (cf. §5.3.3).

Incidently, the issue of generalized case-analysis marks a key philosophical difference of DECLARE compared to Isar. DECLARE essentially takes a few all-inclusive primitives as a starting point, defining simpler concepts as particular instances. In Isar we have started with a generic interpretational model of pure natural deduction proofs, and have built up a hierarchy of derived elements from the very bottom of logical foundations. In moving upwards in the hierarchy of concepts, we have been able to preserve the logical foundations, the operational model of incremental proof processing, and full compositionality with the previous collection of language elements. In fact, this is the deeper reason why Isar turns out as a versatile formal reasoning environment. It has not been *made* to achieve a limited set of goals, but has been *grown* from a sound basis to cover a large field of applications.

In conclusion we propose a slightly different pattern of case-splitting in Isar,

based on quite primitive means of forward-reasoning with proof blocks. The idea is to collect a number of individual results that are integrated by means of a "magical" proof method that needs to take care of eliminating the ultimate covering statement. This is not a first-class context element as **obtain-cases** considered before.

{
  fix $\vec{x}$ **assume** $A_1\ \vec{x}$ **and** $A_2\ \vec{x}$
  hence C $\langle proof \rangle$
} **note** $1 = this$
{
  fix $\vec{y}$ **assume** $B_1\ \vec{y}$ **and** $B_2\ \vec{y}$
  hence C $\langle proof \rangle$
} **note** $2 = this$
from 1 **and** 2 **have** C **by** *blast* — "magical" integration performed here

In fact, this is happens to be an instance of the "big-step" reasoning paradigm performed via automated reasoning tools, see §6.4.3 for slightly more convenient expressions using "degenerate calculations" in Isar (avoiding explicitly named facts like 1 and 2 above). The patterns of "degenerate calculations" presented in §6.4.3 turn out as a fair replacement for the relatively infrequent situations of fully general logical case-splitting encountered in practice. So the lack of real **obtain-cases** in Isar is actually a rather small price for the flexible linear interpretation model of generic natural deduction proof texts (§3.2.3).

Chapter 6

Calculational reasoning

We consider a rather general notion of calculational reasoning, in the sense of iterated equalities and similar relations given in the proof text. Despite being centered around natural deduction, Isar turns out to support a multitude of calculational patterns very well. We do not require any changes to the core concepts of Isar proof processing, but only a few derived elements added on top of the existing framework.

The flexible and non-intrusive manner that calculational reasoning is incorporated into Isar allows free combination with existing natural deduction techniques. This also demonstrates that the two proof styles need be in conflict, but may benefit from each other.

6.1 Introduction

Calculational reasoning essentially proceeds by forming a chain of intermediate results that are meant to be composed by basic principles, such as transitivity of $=/</\leq$ (or similar relations). More advanced calculations may involve substitution, which in the case of inequalities usually includes monotonicity constraints. In informal mathematics, this kind of proof technique is routinely used in a very casual manner. Whenever mathematicians write down sequences of mixed equalities or inequalities, underline subexpressions to be replaced etc., then it is very likely that they are actually doing calculational reasoning.

In fact, calculational reasoning has been occasionally proposed as simple means to rephrase mathematical proof into a slightly more formal setting (e.g. [Back and von Wright, 1999] [Back *et al.*, 1997]), although this does not necessarily include machine-checking of proofs. Observing that logical equivalence and implication may be just as well used in calculations, some have even set out to do away with traditional natural-deduction reasoning altogether (e.g. [Dijkstra

and Scholten, 1990]). The resulting discipline of writing down mathematical proofs in a slightly formalistic manner has been found quite appealing by a considerable number of people, albeit not by everyone.

Nevertheless, calculational reasoning offers a relatively simple conceptual basis to build tools for logical manipulations. For example, the popular Math∫pad tool supports transformations of algebraic expressions in a systematic way. Math∫pad has also acquired means for formal proof checking recently [Verhoeven and Backhouse, 1999], using PVS [Owre *et al.*, 1996] as the backend.

Presently, we cover quite general concepts of calculational reasoning within the Isar framework. We shall see how calculational reasoning may be expressed on top of existing Isar concepts in a very natural manner, and how it may be used in common proof patterns.

The most basic form of calculation in Isar is that of a transitive chain of equalities laid out as follows.

have $x_1 = x_2$ $\langle proof \rangle$
also have $\ldots = x_3$ $\langle proof \rangle$
also have $\ldots = x_4$ $\langle proof \rangle$
finally have $x_1 = x_4$ **.**

According to the general philosophy of Isar, there is *no fixed scheme* for calculations implemented in an ad-hoc fashion. Instead we merely introduce a few derived proof commands and abbreviations (notably "..."), in order to arrive at very general calculational concepts that may be freely combined with the existing natural-deduction proof language. Thus the previous example merely turns out as a idiomatic expression within a larger space of possible expressions.

In particular, Isar calculations may be easily combined with "real" natural deduction elements, without having to subscribe to a fully calculational view of logic in general (as proposed in [Dijkstra and Scholten, 1990]). As we shall point out in further detail later on, there is no need to see calculational reasoning in conflict with traditional natural deduction. Both proof styles are readily available in Isar, leaving the user the free choice of the most appropriate technique in the particular situation at hand.

Speaking in terms of λ-calculus as the canonical model for natural deduction proofs, Isar calculations correspond to "binary" composition of primary and secondary facts, with implicitly determined rules of transitivity. So we exhibit just another concept of formal reasoning that is *in principle* completely redundant, but turns out as indispensable for realistic applications (the same holds for the advanced natural deduction elements covered in chapter 5).

Interestingly, existing tactic-based interactive proof systems such as Isabelle [Paulson and Nipkow, 1994], HOL [Gordon and Melham, 1993], Coq [Barras *et al.*, 1999], PVS [Owre *et al.*, 1996] lack immediate support for calculational reasoning altogether. Even the most basic form of transitive chain given above is very cumbersome to achieve via tactical reasoning.

This omission has been addressed several times in the past. [Simons, 1996] [Simons, 1997] covers specific proof tools to support calculational reasoning within Isabelle tactic scripts. [Grundy, 1991] provides an even more general transformational infrastructure for "window inference". Harrison's "Mizar mode for HOL" simulates a number of concepts of declarative theorem proving on top of the tactic-based HOL-Light system [Harrison, 1996b], including calculational reasoning for mixed transitivity rules.

In formalized mathematical proof, calculations have been recognized as an important concept long ago: Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] supports a fixed format for iterated equations, with implicit application of both transitivity and general substitution rules. [Zammit, 1999a] [Zammit, 1999b] outlines a slightly more flexible version of Mizar calculations for the "SPL" language.

6.2 Foundations of calculational reasoning

6.2.1 Calculational sequences

From a syntactical point of view, the essence of a calculational proof in Isar is that of a *calculational sequence*: let the set *calculation* be freely generated by the constructors *start*: $\mathit{fact} \to \mathit{calculation}$ and *continue*: $\mathit{calculation} \to \mathit{fact} \to \mathit{calculation}$. Apparently, any *calculation* represents a non-empty list of facts. We fine-tune our notation and write canonical calculational sequences $\mathit{continue}\ (\ldots\ (\mathit{continue}\ (\mathit{start}\ a_1)\ a_2)\ \ldots\ a_n)$ concisely as $a_1 \circ a_2 \cdots \circ a_n$, by suppressing *start* and using left-associative infix notation $\circ$ for *continue*.

An *interpreted calculational sequence* shall be any *result* achieved by mapping the *start* and *continue* constructors in a primitive recursive fashion. We only consider interpretations of *calculation* back into *fact*, i.e. result: $\mathit{calculation} \to \mathit{fact}$; we also fix $\mathit{result}\ (\mathit{start}\ a) = a$. There is now only one degree of freedom left to specify the general case of $\mathit{result}\ (c \circ a)$.

The following two kinds of calculational steps will be considered within the Isar framework:

(rule-step): specify $\mathit{result}\ (c \circ a) = r \cdot (\mathit{result}\ c\ @\ a)$ where r is a suitable rule taken from a given context $\mathcal{T}$ of *transitivity rules*. Thus we produce a singleton fact by applying a rule to the current calculational result taken together with some new facts.

(accumulation-step): specify $\mathit{result}\ (c \circ a) = \mathit{result}\ c\ @\ a$, i.e. collect further facts without applying any rule yet.

As a basic example of interpreted calculation sequences, fix the singleton set $\mathcal{T} = \{\vdash a = b \Longrightarrow b = c \Longrightarrow a = c\}$ of transitivities and only perform rule steps; we get $\mathit{result}\ (\vdash x_1 = x_2 \circ \vdash x_2 = x_3 \circ \vdash x_3 = x_4) = \vdash x_1 = x_4$. Thus

we may represent canonical chains of equations composed by plain transitivity. Alternatively, only perform accumulation steps to achieve *result* ($\vdash A_1 \circ \vdash A_2 \circ \vdash A_3$) = [$\vdash A_1$, $\vdash A_2$, $\vdash A_3$], i.e. get a number of facts collected as a separate list. As we shall see later on, even the latter case of seemingly degenerate calculational sequences turns out to be quite useful in practice.

6.2.2 Calculational elements within the proof language

In the next stage we investigate how to provide a proof language interface for the user to compose calculational sequences.

At first sight, the way taken by Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] seems to be the obvious one: simply invent concrete syntax for the primitive operations of composing calculational sequences, and make the implementation support this directly — probably with some link to the basic mechanisms of stating and proving facts. On the other hand, such a way of "making a system do something particular" has its inherent limitations. Typically the yield of a certain amount of implementation effort is just the kind of specific feature one had in mind — no more, no less.

In Isabelle/Isar we follow a different path. Instead of hardwiring calculational reasoning into the basic language, we step back from the immediate implementation problem and figure out how the process of composing calculational sequences may be mapped into the natural flow of reasoning in a non-intrusive fashion. Then by adding only a few abbreviations and conventions, we shall achieve a very general framework for calculational reasoning within Isar requiring only minimal effort. Since this extension happily coexists with the remaining Isar proof language, the resulting space of combined proof patterns contains a large number of practically useful idioms, as shall be explored later on.

How do we actually map calculational sequences into the Isar language? First of all, let us fix a special facts register called "*calculation*" to hold the current state of the (partially interpreted) sequence the user is currently working at. The start of a calculation shall be determined implicitly, as indicated by *calculation* being empty. Whenever a calculation is finished, *calculation* will be reset to await the next sequence to start. The result of a finished sequence is exhibited to the subsequent goal statement as a chained fact; its use in the pending proof is no longer controlled by the calculational process.

Furthermore, we wish to exploit Isar's inherent block structure to support nested calculations. To this end, any update operation on *calculation* shall track the current nesting level, in order to commence a new sequence whenever a new block has been entered. Thus any number of calculational sequences may coexist at different levels of block structure.

We now define derived Isar proof commands **also**, **finally**, **moreover**, and **ultimately** to maintain the *calculation* register as outlined above. The above treatment of block structure is left implicit here.

also	=	**note** *calculation* = *this*	initial
also	=	**note** *calculation* = $r \cdot$ (*calculation* @ *this*)	for $r \in \mathcal{T}$
finally	=	**also from** *calculation*	
moreover	=	**note** *calculation* = *calculation* @ *this*	
ultimately	=	**moreover from** *calculation*	

Here the two main elements are **also** and **moreover**, corresponding to the "rule-steps" and "accumulate-steps" introduced before. The accompanied variants **finally** and **ultimately** finish the current sequence after performing a final step, and exhibit the result. Due to the forward chaining involved in the **from** operation, the next command has to be a goal statement, such as **have** or **show** (cf. the Isar language semantics given in §3.2.3).

This slightly peculiar definition of derived Isar proof commands is essentially sufficient to support calculational reasoning. Only one additional tweak is required in practice, namely the special term binding "..." that refers to the argument term of the most recent explicit fact statement (the argument of a curried infix expression *x op y* refers to its right-hand side *y*). This enables the user to refer to the most relevant bits of the previous calculational statement succinctly. Note that a similar element has already been present in Harrison's "Mizar mode for HOL" [Harrison, 1996b], while actual Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] uses "`.=`" as a special notational device.

We may now rephrase the two basic examples of calculational sequences given in §6.2.1 within the Isar language.

have $x_1 = x_2$ ⟨*proof*⟩
also have $\ldots = x_3$ ⟨*proof*⟩
also have $\ldots = x_4$ ⟨*proof*⟩
finally have $x_1 = x_4$ **.**

have A_1 ⟨*proof*⟩
moreover have A_2 ⟨*proof*⟩
moreover have A_3 ⟨*proof*⟩
ultimately have A_4 **by** (*rule r*)

The primitive notion of calculational sequences does not include the particular manner that actual results get used eventually. The first calculation above employs the most basic pattern of an immediate proof "**.**" (single dot): the final result $\vdash x_1 = x_4$ is applied directly to the claim "**have** $x_1 = x_4$". The second form happens to use the ultimate list of facts to prove a different result by some rule $r = \vdash A_1 \Longrightarrow A_2 \Longrightarrow A_3 \Longrightarrow A_4$.

Expanding the definitions of the derived calculational commands introduced before, we may achieve the following raw Isar proofs (here we refer to concrete attribute syntax for composition of facts, cf. §3.3.2).

have $x_1 = x_2$ ⟨*proof*⟩
note *calculation* = *this*
— first rule step: init *calculation* register

have ... = x_3 ⟨*proof*⟩
note *calculation* = *trans* [*OF calculation this*]
— general rule step: compose with transitivity rule
have ... = x_4 ⟨*proof*⟩
note *calculation* = *trans* [*OF calculation this*]
— final rule step: compose with transitivity rule ...
from *calculation*
— ... and pick up the result
have $x_1 = x_4$ **.**

note *calculation* = *nothing*
— purge *calculation* register before commencing next sequence

have A_1 ⟨*proof*⟩
note *calculation* = *calculation this*
— general accumulation step: collect fact
have A_2 ⟨*proof*⟩
note *calculation* = *calculation this*
— general accumulation step: collect fact
have A_3 ⟨*proof*⟩
note *calculation* = *calculation this*
— ultimate accumulation step: collect fact ...
from *calculation*
— ... and pick up the result
have A_4 **by** (*rule r*)

Certainly, the composition of the underlying primitive sequences of facts may be simulated by "pure" natural deduction techniques as well, involving only backwards reasoning. Thus we would get the following versions instead.

have $x_1 = x_4$
proof (*rule trans*)
show $x_1 = x_3$
proof (*rule trans*)
show $x_1 = x_2$ ⟨*proof*⟩
show $x_2 = x_3$ ⟨*proof*⟩
qed
show $x_3 = x_4$ ⟨*proof*⟩
qed

have A_4
proof (*rule r*)
show A_1 ⟨*proof*⟩
show A_2 ⟨*proof*⟩
show A_3 ⟨*proof*⟩
qed

The first proof above exhibits all of the cumbersome formal detail required to compose chains of basic facts naively. This involves explicit rule applications of $trans = \vdash a = b \Longrightarrow b = c \Longrightarrow a = c$, nested sub-proofs, and repeated intermediate statements. Traditional tactic scripts would usually proceed along the same line, although that slightly awkward procedure would typically not be visible from the code.

Our second backwards proof looks quite handsome at first sight, apart from the separately nested sub-proof (which may occasionally become slightly cumbersome as well). Plain back-chaining of a rule r is perfectly adequate in many situations. On the other hand, there are useful patterns of the original calculational version, particularly in conjunction with automated proof tools to process the accumulated result (see also §6.4.3).

6.2.3 Rules and proof search

An important design philosophy of Isar is to keep automated proof tools separate from the key mechanisms of interpreting high-level proof texts. Only linear search over a limited number of possibilities plus (higher-order) unification will be accepted here. The place of complex automated proof methods is usually in terminal positions, to justify local claims in the context of certain facts in isolation (see also §7.5.2).

Reconsidering the commands for outlining calculational sequences (§6.2.2), we see that there is a single non-deterministic parameter, namely the rule $r \in \mathcal{T}$ to be selected by the general **also** element. As Isar proof texts are interpreted strictly from left to right (§3.2.3), any subsequent result $calculation = r \cdot (calculation \mathbin{@} this)$ has to be achieved from the present facts alone, with the rule instance r determined by the system appropriately. As long as $\mathcal{T}$ only holds canonical transitivities of $=/</\leq$ the result is already uniquely determined, e.g. providing facts $\vdash x = y$ and $\vdash y = z$ invariably yields $\vdash x = z$.

Isar uses the following refined strategy to support more general rule selections. Assume a canonical order on the rule context $\mathcal{T}$, and let $a = calculation \mathbin{@} this$ be the input given to the present calculational step. Now enumerate the members r of $\mathcal{T}$ according to their priority, then enumerate the canonical sequences of results $r \cdot a$ as obtained by parallel higher-order unification and back-chaining of a with r (cf. §2.4). Finally filter this raw result sequence to disallow mere projections of a; in other words remove those results b that do not make any "progress", in the sense that the conclusion of b is already present in one of the members of the list a.

This strategy subsumes the simple case of unique results considered before, but also does a good job at substitution: let us declare $\vdash P\ x \Longrightarrow x = y \Longrightarrow P\ y$ and $\vdash y = x \Longrightarrow P\ x \Longrightarrow P\ y$ to be tried after the plain transitivities considered so far. The expression $x = y$ only requires plain first-order unification, with a unique most-general result. The critical part is to solve $P\ x$ against the other expression provided in the calculation, which is genuine higher-order problem.

The resulting unifiers will assign a certain λ-term to P that abstracts over possible occurrences of sub-expression x. Here the standard strategy [Paulson, 1986] is to start with a solution with all occurrences, followed by all possible partial occurrences in a fixed order, and finish with *no* occurrences. Note that the latter case is the only possible solution if x does not occur at all, which is actually a pathological case for our purpose, since it collapses the substitution rules to $\vdash P \Longrightarrow x = y \Longrightarrow P$ and $\vdash y = x \Longrightarrow P \Longrightarrow P$, respectively.

Thus by filtering out mere projections like this, a basic calculational rule-step is able to produce a sensible result, where *all* occurrences of a certain sub-expression may be replaced by an equal one. Replacing only *some* occurrences does not work, though, as there is no way in Isar to specify the intended result of a calculational step directly. In practice, ill-behaved substitutions are usually better replaced by plain transitivity, mentioning the full term context explicitly in the text and making the justification step take care of normalizing the claim appropriately (e.g. by Isabelle's Simplifier [Nipkow *et al.*, 2001], see also §7.3).

Substitution by greater (or equal) sub-expressions with additional monotonicity constraints works as well, see also §6.3. The only caveat is that our notion of "progress" in the filtering strategy really has to ignore local assumptions, because higher-order resolution would insert additional premises even in the degenerate cases of higher-order unification (§2.4).

6.3 Common patterns of calculational reasoning

The space of possible calculational expressions within Isar is somewhat open-ended, due to the very nature that calculational primitives have been incorporated into the basic proof language. Certainly, creative users of Isabelle/Isar may invent further ways of calculational reasoning at any time. Here we point out possible dimensions of variety, and outline practically useful idiomatic patterns. Our subsequent categories are guided by the way that primitive calculational sequences may be mapped into the Isar proof language, interacting with different categories of existing language elements.

6.3.1 Variation of rules

Mixed transitivity

The most basic form of calculation is a plain transitive chain of equations, as we have already encountered before. Mixed transitivities may be used as follows: observe that the canonical ending (with a single-dot proof) forces the final goal statement to exhibit the result explicitly in the text.

have $x_1 \leq x_2$ $\langle proof \rangle$
also have $\ldots \leq x_3$ $\langle proof \rangle$
also have $\ldots = x_4$ $\langle proof \rangle$

also have $\ldots < x_5$ ⟨*proof*⟩
also have $\ldots = x_6$ ⟨*proof*⟩
finally have $x_1 < x_6$ **.**

Likewise, we may use further combinations of relations like antisymmetry, as long as there is a clear functional mapping from facts to the result and no conflict with other rules.

have $x \leq y$ ⟨*proof*⟩
also have $y \leq x$ ⟨*proof*⟩
finally have $x = y$ **.**

Substitution

The technical caveats of calculating with substitution rules have already been covered in §6.2.3. The fine-tuned filtering of rule selections discussed before admits *consistent* replacement of equals by equals without further ado, as illustrated below.

have $A = B + f\, x + C + x$ ⟨*proof*⟩
also have $x = y$ ⟨*proof*⟩
also have $B + f\ \ldots + C + \ldots = D$ ⟨*proof*⟩
finally have $A = D$ **.**

In practice, calculations mostly consist of plain transitive steps with only very few substitutions interspersed. These may be easily spotted in the text according to the following discipline: the replacement statement ($x = y$ above) mentions proper sub-terms of the previous stage (without referencing "..."), while in the consecutive stage the right-hand side of the replacement is *documented* by appropriate occurrences of "..." in the result (reconsider $B + f\, x + C + x$ versus $B + f\ \ldots + C + \ldots = D$ above).

Substitution with inequalities essentially works in a similar fashion, although the rules need to be formulated slightly more specifically and include a separate monotonicity condition. Consider the following example.

have $A = B + x + C$ ⟨*proof*⟩
also have $x \leq y$ ⟨*proof*⟩
also have $B + \ldots + C = D$ ⟨*proof*⟩
finally have $A \leq D$
proof *this*
  fix $u\ v$ **assume** $u \leq v$
  thus $B + u + C \leq B + v + C$ **by** *simp*
qed

The rule used here is $\vdash a = f\, b \Longrightarrow b \leq c \Longrightarrow (\bigwedge u\ v.\ u \leq v \Longrightarrow f\, u \leq f\, v) \Longrightarrow a \leq f\, c$, which has three premises, but we have only filled in the first two facts in the calculation; the remaining monotonicity constraint has been left as additional hypothesis of the result, which eventually was solved by hand in the proof annex given above.

Incidently, the monotonicity constraint already gets fully instantiated by giving the first two facts only; thus the remainder is usually rather easily proven by common automated tools. So in reality we would usually collapse the final sub-proof to "**by** *this simp*", or even just "**by** *simp*". We see how high-level calculational proof outlining nicely works hand-in-hand with dumb automation.

have $A = B + x + C$ $\langle proof \rangle$
also have $x \leq y$ $\langle proof \rangle$
also have $B + \ldots + C = D$ $\langle proof \rangle$
finally have $A \leq D$ **by** *simp*

In some cases, one may want to provide all three premises directly. This is easily achieved by using **moreover** and **also** in combination, accumulating any additional facts before the calculational rule is fired.

For example, cf. the Knaster-Tarski proof in §1.5 with the characteristic phrase of "**moreover note** *mono*". Slightly more realistic applications of calculations with monotonicity constraints are given in [Bauer and Wenzel, 2001] [Bauer, 2001b], covering a set-theoretic model of Computational Tree Logic (CTL).

Modus ponens

We may also calculate directly with logical propositions, getting somewhat closer to the proof style of [Dijkstra and Scholten, 1990]. The following pattern essentially achieves "light-weight" natural deduction, by implicit use of the *modus ponens* rule.

have $A \longrightarrow B \longrightarrow C$ $\langle proof \rangle$
also have A $\langle proof \rangle$
also have B $\langle proof \rangle$
finally have C **.**

In principle, transitivity of "$\longrightarrow$" may be as a valid calculational rule as well, although chaining of implications is more conveniently expressed directly by Isar's **then** primitive circumventing the overhead of explicit logical connectives altogether. See also §6.4.2 for further issues of calculating with propositions.

Rules of Isabelle/HOL

Finally we present the collection of standard calculational rules as included in the main Isabelle/HOL library (see also chapter 7).

$\vdash a = b \Longrightarrow b = c \Longrightarrow a = c$
$\vdash a = b \Longrightarrow b < c \Longrightarrow a < c$
$\vdash a < b \Longrightarrow b = c \Longrightarrow a < c$
$\vdash a = b \Longrightarrow b \leq c \Longrightarrow a \leq c$

$\vdash a \leq b \Longrightarrow b = c \Longrightarrow a \leq c$
$\vdash a \leq b \Longrightarrow b \leq a \Longrightarrow a = b$
$\vdash a \leq b \Longrightarrow b \leq c \Longrightarrow a \leq c$
$\vdash a < b \Longrightarrow b \leq c \Longrightarrow a < c$
$\vdash a \leq b \Longrightarrow b < c \Longrightarrow a < c$
$\vdash a < b \Longrightarrow b < a \Longrightarrow C$
$\vdash a < b \Longrightarrow b < c \Longrightarrow a < c$
$\vdash a \leq b \Longrightarrow a \neq b \Longrightarrow a < b$
$\vdash a \neq b \Longrightarrow a \leq b \Longrightarrow a < b$
$\vdash A \subseteq B \Longrightarrow x \in A \Longrightarrow x \in B$
$\vdash x \in A \Longrightarrow A \subseteq B \Longrightarrow x \in B$
$\vdash A \longrightarrow B \Longrightarrow A \Longrightarrow B$
$\vdash A \Longrightarrow A \longrightarrow B \Longrightarrow B$
$\vdash P\ a \Longrightarrow a = b \Longrightarrow P\ b$
$\vdash a = b \Longrightarrow P\ b \Longrightarrow P\ a$
$\vdash a = f\ b \Longrightarrow b < c \Longrightarrow (\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow a < f\ c$
$\vdash a < b \Longrightarrow f\ b = c \Longrightarrow (\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow f\ a < c$
$\vdash a = f\ b \Longrightarrow b \leq c \Longrightarrow (\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow a \leq f\ c$
$\vdash a \leq b \Longrightarrow f\ b = c \Longrightarrow (\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow f\ a \leq c$
$\vdash a \leq f\ b \Longrightarrow b \leq c \Longrightarrow (\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow a \leq f\ c$
$\vdash a \leq b \Longrightarrow f\ b \leq c \Longrightarrow (\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow f\ a \leq c$
$\vdash a < f\ b \Longrightarrow b \leq c \Longrightarrow (\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow a < f\ c$
$\vdash a < b \Longrightarrow f\ b \leq c \Longrightarrow (\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow f\ a < c$
$\vdash a \leq f\ b \Longrightarrow b < c \Longrightarrow (\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow a < f\ c$
$\vdash a \leq b \Longrightarrow f\ b < c \Longrightarrow (\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow f\ a < c$
$\vdash a < f\ b \Longrightarrow b < c \Longrightarrow (\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow a < f\ c$
$\vdash a < b \Longrightarrow f\ b < c \Longrightarrow (\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow f\ a < c$

Note that meaningful calculations may already be performed with basic transitivity of "=" alone. So users who start new object-logics may be content to begin with significantly fewer rules (see also the primitive formulation of basic higher-order logic in chapter 8). Even the degenerate version without using any rules has its distinctive applications (see also §6.4.3).

In any case, end-users may easily declare further transitivity rules for specific relations occurring in the particular applications at hand. Concrete syntax to manipulate the implicit rule context $\mathcal{T}$ is provided by the *trans* attribute (see also [Wenzel, 2001a]).

6.3.2 Variation of conclusions

Recall that the business of managing the calculational process actually finishes with the concluding **finally** or **ultimately** command, which exhibits the result with forward-chaining indicated (cf. §6.2.2). The next command needs to be a plain goal of the Isar language, such as **have** or **show** (cf. §3.2.1). The most basic proof of such a claim is just "**.**", meaning that the goal statement actually

reiterates the calculational result directly (or a substitution instance).

Occasionally, one might even wish to modify the final result via a single canonical rule that is easy to oversee, e.g. symmetry of "=" as illustrated below.

have $x_1 = x_2$ $\langle proof \rangle$
also have $\ldots = x_3$ $\langle proof \rangle$
also have $\ldots = x_4$ $\langle proof \rangle$
finally have $x_4 = x_1$ **..**

Another useful idiom is to feed the result, which may be just a number accumulated facts, into a single rule with several premises. This technique is illustrated by the following forward-proof of $A \wedge B \longrightarrow B \wedge A$.

lemma $A \wedge B \longrightarrow B \wedge A$
proof
assume ab: $A \wedge B$
hence B **..**
moreover
from ab **have** A **..**
ultimately
show $B \wedge A$ **..**
qed

Here the result emerges in the ultimate "**..**" proof by $\wedge$ introduction. This pattern may be easily generalized to any number of premises and arbitrary proof methods, eventually achieving a version of "big-step" inferences (see also §6.4.3).

Certainly, the **obtain** element (§5.3) may be used as a concluding goal statement as well, being based on plain **have** internally. Subsequently we give a slightly unusual calculation, consisting of mere accumulation steps with the ultimate result being used obtain the very same facts simultaneously.

have A $\langle proof \rangle$
moreover have B $\langle proof \rangle$
moreover have C $\langle proof \rangle$
ultimately obtain A **and** B **and** C **..**

Note that the ultimate proof needs to perform a single step (via "**..**") due to the standard introduction pattern of **obtain** (cf. §5.3.3).

6.3.3 Variation of facts

In all calculational schemes encountered so far, the facts placed into the chain have been produced by intermediate statements "**have** φ $\langle proof \rangle$". This happens to be the most common pattern in practice, but does not constitute an inherent restriction. Any other Isar language element that yields a result may be used in calculations as well. This includes **note** (§3.2.1) to recall existing

theorems, or goal elements such as **show** (§3.2.1), or even context commands such as **assume** (§3.3.1), or **obtain** (§5.3).

For example, the Knaster-Tarski proof in §1.5 illustrates very basic use of "**moreover note**" to include an auxiliary fact into the calculational sequence; "**also note**" frequently occurs in many applications as well. Combinations with **obtain** are very useful in typical computer-science applications involving abstract syntactic models (e.g. see chapter 10), but also in classical mathematics involving representation proofs (as in chapter 9).

The use of **assume** within a calculation represents the most basic case of combining calculational reasoning and pure natural deduction. Consider the following induction proof of the summation formula for odd numbers.

In our first version below there is no tight integration of the two styles of reasoning, yet. We use naive backwards reasoning at the outer level, with a separate local calculation to establish the induction step.

lemma $(\sum k < n.\ 2 * k + 1) = n^2$ (**is** *?S n* = -)
proof (*induct n*)
  show *?S* $0 = 0^2$ **by** *simp*
next
  fix *n* **assume** *hyp*: *?S n* $= n^2$
  show *?S* (*Suc n*) $= (Suc\ n)^2$
  proof −
    have *?S* (*Suc n*) $= 2 * n + 1 +$ *?S n* **by** *simp*
    also note *hyp*
    also have $2 * n + 1 + n^2 = (Suc\ n)^2$ **by** *simp*
    finally show *?thesis* **.**
  qed
qed

A slightly less formalistic proof may be achieved by a number of straight-forward rearrangements: defer the main inductive goal to the very end (casual forward reasoning instead of strictly hierarchical organization); introduce the induction hypothesis at the place where it is actually used (possible due to flat structure); eliminate superfluous naming of assumption (due to natural flow of facts); use $n + 1$ instead of *Suc n* in local statements (due to "**by** *simp*" in the final step).

lemma $(\sum k < n.\ 2 * k + 1) = n^2$ (**is** *?S n* = -)
proof (*induct n*)
  show *?S* $0 = 0^2$ **by** *simp*
next
  fix *n* **have** *?S* $(n + 1) = 2 * n + 1 +$ *?S n* **by** *simp*
  also assume *?S n* $= n^2$
  also have $2 * n + 1 + \ldots = (n + 1)^2$ **by** *simp*
  finally show *?S* (*Suc n*) $= (Suc\ n)^2$ **by** *simp*
qed

The deeper reason why natural deduction elements (namely assumptions) may

get introduced in the middle of calculational sequences is the way that Isar proof contexts are managed, completely independently of any goal statements (cf. §3.3.1 and §5.2.1). In particular, the proof context is inherently "cascaded", in the sense that the scope of conceptual λ-abstractions introduced via the **assm** primitive (and its derived forms like **assume**, cf. §3.3.1) spans over the remaining proof body, until the next closing block (cf. §5.2.1).

6.3.4 Variation of general structure

Calculational sequences are basically linear, but arbitrary many intermediate steps may be taken until the next fact is produced. This may include further nested calculations, as long as these are arranged on separate levels of block structure. Nested calculations routinely emerge by virtue of the implicit block structure of sub-proofs. Raw proof blocks may be used as well (cf. §5.2.4), although this occurs less frequently in practice. The subsequent pattern illustrates both instances.

have $x_1 = x_2$ $\langle proof \rangle$
also have $\ldots = x_3$
proof −
  have $\ldots = y_1$ $\langle proof \rangle$
  also have $\ldots = y_2$ $\langle proof \rangle$
  also have $\ldots = x_3$ $\langle proof \rangle$
  finally show *?thesis* **.**
qed
also {
  have $\ldots = z_1$ $\langle proof \rangle$
  also have $\ldots = z_2$ $\langle proof \rangle$
  also have $\ldots = x_4$ $\langle proof \rangle$
  finally have $x_3 = x_4$ **.**
}
finally have $x_1 = x_4$ **.**

Appropriate use of nested calculations is left to the discernment of the user (cf. the general liberality principle of the Isar, §1.3),

6.4 Discussion

6.4.1 Iterated equalities in Mizar

Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] has focused on formal proof in common mathematics style from its very beginnings. Consequently, it also offers a mechanism for iterated equality reasoning according to basic calculations seen in typical mathematical reasoning.

Iterated equalities are hard-wired into the Mizar implementation. Composition of individual steps may involve transitivity or substitution of the equality relation (over individuals of first-order logic). The following trivial example is taken from article #185 of [Mizar library].

```
theorem Th1:
  for X,Y being set holds union {X,Y,{}} = union {X,Y}
proof
   let X,Y be set;
   thus union {X,Y,{}} = union ({X,Y} U {{}}) by ENUMSET1:43
                .= union {X,Y} U union {{}} by ZFMISC_1:96
                .= union {X,Y} U {} by ZFMISC_1:31
                .= union {X,Y};
end;
```

Recall that `thus` in Mizar indicates that the subsequent statement is meant to solve a pending goal. Furthermore, the "continued equality" sign "`.=`" indicates that the actual result shall emerge from a number of individual equations, where transitivity steps are handled behind the scenes.

The very same example works out in Isabelle/Isar as follows, at the mercy of automated reasoning tools available in Isabelle (see also §7.3).

theorem $\bigcup\{X,\ Y,\ \{\}\} = \bigcup\{X,\ Y\}$
  by *auto*

Similar proof tools of HOL [Gordon and Melham, 1993] or PVS [Owre *et al.*, 1996] would certainly solve such trivial problems without further ado as well.

Indeed, many calculations in the Mizar library are of the same kind as above. This incident indicates that Mizar's builtin proof tools do not handle equality too well; there is no direct support for rewriting. Consequently, basic simplifications need to be performed by hand via numerous steps of iterated equalities. Nevertheless, iterated equalities in Mizar turn out as an indispensable means for proper arrangement of many practical relevant proof patterns. Obviously, arbitrary automated reasoning is *not* a replacement for proper concepts of high-level proof structure.

The next version takes the business of calculational reasoning in Isar seriously. We mimic the original Mizar proof as closely as possible.

theorem $\bigcup\{X,\ Y,\ \{\}\} = \bigcup\{X,\ Y\}$
proof −
  have $\bigcup\{X,\ Y,\ \{\}\} = \bigcup(\{X,\ Y\} \cup \{\{\}\})$ **by** *auto*
  also have $\ldots = \bigcup\{X,\ Y\} \cup \bigcup\{\{\}\}$ **by** *auto*
  also have $\ldots = \bigcup\{X,\ Y\} \cup \{\}$ **by** *auto*
  also have $\ldots = \bigcup\{X,\ Y\}$ **by** *auto*
  finally show *?thesis* **.**
qed

It is important to note that this enlarged text is still meaningful, despite being rather redundant in terms of the automated proof method available in Isabelle.

Each terminal sub-proof of "**by** *auto*" is strictly limited to a local problem, there is no way to disturb the global arrangement of reasoning accidentally. The calculational sequence is evaluated independently of the intermediate steps taken in the proof process.

This robustness of proof texts against arbitrary operations performed inside of proof methods is essentially a consequence of modularity of Isar proof processing (cf. chapter 3).

Comparing the overall structure of Isar calculations with iterated equalities in Mizar, we first observe that Mizar does not require separate language elements to manage the course of reasoning (unlike **also**/**finally** in Isar). Mizar is content with "`.= y`" where Isar would typically require "**also have** $\ldots = y$" to be spelled out explicitly. On the other hand, the latter just happens to be a particular idiomatic expression within a more general playground (cf. §6.3).

Another notable difference is how the final result is exhibited: in Mizar it is made directly available to the proof context (here as the result of `thus`, which solves the main goal). In Isar the calculational result is always indicated for forward chaining, which forces the next command to be a goal statement that is meant to use that fact in an appropriate manner. The most common Isar idiom is to reiterate the result in the text and prove it immediately (via "**.**"), as in "**finally show** φ **.**" for a result of $\vdash \varphi$. This scheme of Isar avoids "magical" appearance of implicit facts from calculations spanning several lines (or even pages as routinely encountered in the Mizar library).

Looking more closely how the previous calculational proof actually proceeds, we see that there are a number of substitution steps involved, although the top-level chain has been presented as a plain transitive one by repeating previous term contexts. Both Isar and Mizar are able to perform substitutions directly as well. The subsequent Isar proof makes this explicit, where we use the generic "..." term abbreviation to indicate the positions of replacement according to the substitution discipline outlined in §6.3.1.

theorem $\bigcup\{X,\ Y,\ \{\}\} = \bigcup\{X,\ Y\}$
proof −
  have $\{X,\ Y,\ \{\}\} = \{X,\ Y\} \cup \{\{\}\}$ **by** *auto*
  also have $\bigcup \ldots = \bigcup\{X,\ Y\} \cup \bigcup\{\{\}\}$ **by** *auto*
  also have $\bigcup\{\{\}\} = \{\}$ **by** *auto*
  also have $\bigcup\{X,\ Y\} \cup \ldots = \bigcup\{X,\ Y\}$ **by** *auto*
  finally show *?thesis* **.**
qed

It is important to note that the above use of "..." does *not control* substitution steps, but only *documents* its effect. In contrast, Mizar is bound to the fixed "`.=`" notation even in substitution steps; thus non-trivial chains are somewhat harder to read than in Isar. Nevertheless, substitution chains should not be made overly smart. Many practical applications are more adequately represented by plain transitivity, leaving the details of substitution to the justification

steps (e.g. to the *simp* or *auto* proof methods of Isabelle, see also §7.3).

In conclusion, Mizar's concept of iterated equalities may be understood as a hardwired version of a fixed format of Isar calculations, picking the following parameters of our framework (cf. §6.3).

- Individual steps are always linked via **also** using either plain transitivity or substitution (of "=" over individuals of first-order logic only).
- Conclusions may be either "**finally have**" or "**finally show**", although the final result would emerge immediately in the Mizar proof text, without the additional proof step encountered in Isar.
- Intermediate facts always emerge from "**have** φ $\langle$*proof*$\rangle$" elements.
- The "..." term abbreviation degenerates into a formal device to indicate continued equality (cf. Mizar's special "`.=`" notation).

Apparently, Isar has been able to offer a more flexible calculational environment, with only very few conservative additions to the existing framework. There is no need to hardwire calculational concepts into the existing process of high-level proof processing for natural deduction (chapter 3).

6.4.2 Dijkstra's universal calculational proof format

The calculational proof format proposed by Dijkstra (e.g. [Dijkstra and Scholten, 1990]) has a strong focus on direct manipulation of boolean expressions rather than mere relations over individual elements. The most basic pattern introduced in [Dijkstra and Scholten, 1990, §4] is that of iterated logical equivalence, eventually yielding boolean values of *true* or *false*.

$$
\begin{array}{ll}
 & [A] \\
= & \quad \{\text{hint why } [A] \equiv [B]\} \\
 & [B] \\
= & \quad \{\text{hint why } [B] \equiv [C]\} \\
 & [C] \\
= & \quad \{\text{hint why } [C] \equiv \mathit{true}\} \\
 & \mathit{true}
\end{array}
$$

Note that the square bracket notation "$[A]$" refers to Dijkstra's builtin treatment of implicit state dependency of logical expressions, cf. the discussion of the specific notion of "structures" in [Dijkstra and Scholten, 1990, §1]. [Harrison, 1998] provides a somewhat simpler representation of this basic idea within the language of HOL, using plain abstraction and universal quantification.

Further abbreviations are used for the special case of propositions stating equality of individuals. This essentially results in transitive chains over the equality relation, although Dijkstra requires Leibniz's rule to justify that pattern on top of the previous propositional one [Dijkstra and Scholten, 1990, page 23].

$$\begin{array}{ll} & A \\ = & \quad \{\text{hint why } [A = B]\} \\ & B \\ = & \quad \{\text{hint why } [B = C]\} \\ & C \\ = & \quad \{\text{hint why } [C = D]\} \\ & D \end{array}$$

In the Isar framework such proof patterns could be explained easily via calculational sequences using plain transitivity rules of logical equivalence and equality (cf. §6.2.1). In the HOL logic both rules would even coincide, since there is nothing special about the type *bool* [Church, 1940] [Gordon and Melham, 1993]. Thus we could calculate directly with propositions, as illustrated below. Note that we refer to additional standard elimination rules $\vdash A = \mathit{True} \Longrightarrow A$ and $\vdash A = \mathit{False} \Longrightarrow \neg A$ to streamline the final results.

have $A = B$ ⟨*proof*⟩
also have $\ldots = C$ ⟨*proof*⟩
also have $\ldots = \mathit{True}$ ⟨*proof*⟩
finally have A **..**

have $A = B$ ⟨*proof*⟩
also have $\ldots = C$ ⟨*proof*⟩
also have $\ldots = \mathit{False}$ ⟨*proof*⟩
finally have $\neg A$ **..**

Dijkstra's proof format is intended as a replacement of natural deduction, logical equivalence is generally preferred over directed reasoning from assumptions to conclusions. Furthermore, the "hints" justifying individual calculational steps are drawn from the "calculus of boolean structures" [Dijkstra and Scholten, 1990, §5], featuring a number of specific rules such as the lattice-theoretical fact $[A \wedge B \equiv A \equiv B \equiv A \vee B]$ (called the "Golden Rule").

The example of $A \wedge B \equiv B \wedge A$ shall illustrate the resulting kind of reasoning with boolean structures, cf. [Dijkstra and Scholten, 1990, p. 38].

$$\begin{array}{ll} & A \wedge B \\ = & \quad \{\text{Golden Rule}\} \\ & A \equiv B \equiv A \vee B \\ = & \quad \{\text{associativity and symmetry of } \equiv\} \\ & B \equiv A \equiv A \vee B \\ = & \quad \{\text{symmetry of } \vee\} \\ & B \equiv A \equiv B \vee A \\ = & \quad \{\text{Golden Rule}\} \\ & B \wedge A \end{array}$$

The treatment of full logical equivalence is usually quite handsome in Dijkstra's algebraic setting of boolean structures. Nevertheless, plain implications are

occasionally required in practice whenever only one direction happens to hold, or the other one is hard to prove and not really required. Dijkstra's format allows to mix equivalence with implication (in either direction). Some care has to be taken to direct the calculation properly, depending on the final result of either *true* or *false*, namely $A \Longleftarrow \mathit{true}$ versus $A \Longrightarrow \mathit{false}$. Naturally, the converse statements are useless, although they readily occur in proof attempts of beginners.

In Isar we may easily emulate the mixed form of calculating with transitivity rules for (backwards and forwards) implication on propositions. Again we prefer to simplify final implications involving basic boolean values.

have $A \longleftarrow B$ $\langle \mathit{proof} \rangle$
also have $\ldots = C$ $\langle \mathit{proof} \rangle$
also have $\ldots \longleftarrow \mathit{True}$ $\langle \mathit{proof} \rangle$
finally have A **..**

have $A \longrightarrow B$ $\langle \mathit{proof} \rangle$
also have $\ldots = C$ $\langle \mathit{proof} \rangle$
also have $\ldots \longrightarrow \mathit{False}$ $\langle \mathit{proof} \rangle$
finally have $\neg\ A$ **..**

Subsequently, we expose the complete collection of transitivity rules required to emulate common proof patterns of Dijkstra's calculational format in Isar.

$\vdash A \longrightarrow B \Longrightarrow B \longrightarrow C \Longrightarrow A \longrightarrow C$
$\vdash A = B \Longrightarrow B \longrightarrow C \Longrightarrow A \longrightarrow C$
$\vdash A \longrightarrow B \Longrightarrow B = C \Longrightarrow A \longrightarrow C$
$\vdash A \longleftarrow B \Longrightarrow B \longleftarrow C \Longrightarrow A \longleftarrow C$
$\vdash A = B \Longrightarrow B \longleftarrow C \Longrightarrow A \longleftarrow C$
$\vdash A \longleftarrow B \Longrightarrow B = C \Longrightarrow A \longleftarrow C$

The following rules may be declared as standard eliminations, in order to be able to conclude plain propositions as final result (merely using a single "**..**" proof step, instead of the common "**.**"). Certainly, such formal details would be omitted in an informal setting like the original one of Dijkstra.

$\vdash A \longleftarrow \mathit{True} \Longrightarrow A$
$\vdash A = \mathit{True} \Longrightarrow A$
$\vdash A \longrightarrow \mathit{False} \Longrightarrow \neg\ A$
$\vdash A = \mathit{False} \Longrightarrow \neg\ A$

We see that the high-level natural deduction framework of Isar is able to assimilate Dijkstra's algebraic view on logic quite easily. In fact, the two styles may peacefully coexist within the same environment, enabling the user to choose the preferred techniques according to the particular situation at hand.

On the other hand, the standard Isabelle library does *not* declare the specific collection of propositional calculational rules as presented above. While logical equivalence is already covered by the general "=" relation (in HOL), we have

intentionally omitted any implication rules (apart from plain *modus ponens*, cf. §6.3.1). In fact, serious applications would really demand backward implication "⟵", which is absent in the Isabelle/HOL library in the first place.

The deeper reason for excluding transitive chains of implication by default is that Isar's very natural deduction core is more appropriate to treat this issue directly, without requiring any particular logical connectives.

As a general rule of thumb, "native" Isar language elements should be preferred over any "encoding" of concepts within the logic, since such indirection would have to be taken care of explicitly within proofs. As a notable example for the same principle reconsider the abstract handling of nested "∃" and "∧" forms via the derived Isar element **obtain** (cf. §5.3). The case of iterated (forward) implication is an even more fundamental one, being already inherently present in the basic concept of forward-chaining in Isar (chapter 3), as illustrated below.

{
assume A_1
hence A_2 ⟨*proof*⟩
hence A_3 ⟨*proof*⟩
hence A_4 ⟨*proof*⟩
}
hence $A_1 \longrightarrow A_4$ **..**

In typical applications we would not even need to exhibit the final statement $A_1 \longrightarrow A_4$, the proof may usually just continue with the dependent result in a casual manner.

The following variant calculates with transitivity of implication instead of using native forward-chaining seen before.

have $A_1 \longrightarrow A_2$ ⟨*proof*⟩
also have $\ldots \longrightarrow A_3$ ⟨*proof*⟩
also have $\ldots \longrightarrow A_4$ ⟨*proof*⟩
finally have $A_1 \longrightarrow A_4$ **.**

At first sight, this pattern appears to be quite handsome after all. On the other hand, the individual justification steps need to tackle a slightly different goal statement involving explicit implications. Unless some automated proof support takes care of this behind the scenes, we would have to decompose the statements directly within the text. Since single rule applications tend to be used frequently in Isar, this minor difference may already be considered as one unnecessary complication too many.

The formal noise hidden here is made explicit as follows. After initial decomposition, the remaining justifications are technically exactly the same as in the version with plain forward-chaining given before.

have $A_1 \longrightarrow A_2$
proof
assume A_1

thus A_2 ⟨*proof*⟩
qed
also have ... ⟶ A_3
proof
assume ...
thus A_3 ⟨*proof*⟩
qed
also have ... ⟶ A_4
proof
assume ...
thus A_4 ⟨*proof*⟩
qed
finally have $A_1 \longrightarrow A_4$ **.**

The difference of derivations from facts versus explicit intra-logical implication encountered here would not really matter in Dijkstra's original informal setting. In Isar any additional formal detail is apt to intrude proof texts immediately, so we need to be more careful to avoid unnecessary clutter of formal proof texts.

6.4.3 Degenerate calculations and big-step reasoning

Corresponding to the notions of rule-steps versus accumulation-steps (§6.2.2), Isar provides two kinds of calculational commands, namely **also**/**finally** versus **moreover**/**ultimately** (§6.2.1). Calculations composed by **also**/**finally** may be considered as "proper" ones, corresponding closely to the original idea of iterated applications of (implicit) transitivity rules. Occasional **moreover** steps may get included here as well, in order to accommodate rules with more than two premises (cf. §6.3.1).

Actual "degenerate" calculations would only consist of **moreover** steps and conclude via **ultimately**. Here the effect is to collect a number of facts over a certain range of proof text, and exhibit it as a list of theorems to the ultimate claim. The latter may involve any goal statement, established by an arbitrary proof method (cf. §6.3.2).

Degenerate calculations turn out as a surprisingly useful concept in their own right, mainly due to the following key properties of Isar.

1. Facts contributing to a calculation may consist of *any* proof text that produces results eventually. In particular, this may include explicit blocks "**{** ... **}**" to manage independent local contexts.

2. The ultimate conclusion may involve any proof method, including advanced automated tools.

3. Calculations may be nested according to the block structure of the text.

As already seen earlier (§6.3.2), **moreover**/**ultimately** may essentially be used to invert the course of basic natural deduction. This results in a strongly forward style, without demanding excessive use of explicit references to previous facts in the text. For example, consider the following version of $\wedge$ introduction.

have A ⟨*proof*⟩
moreover have B ⟨*proof*⟩
ultimately have $A \wedge B$ **..**

This scheme may be easily generalized to any number of conjuncts, by using Isabelle's classical tableau prover *blast* (see §7.3) in the ultimate step (this rather trivial case happens to work with plain *simp* as well).

have A_1 ⟨*proof*⟩
moreover have A_2 ⟨*proof*⟩
moreover have A_3 ⟨*proof*⟩
ultimately have $A_1 \wedge A_2 \wedge A_3$ **by** *blast*

The dual pattern of $\vee$ elimination essentially works in the same way (here the "logical" reasoning capabilities of *blast* will be really required). The facts contributing to the calculation emerge from proof blocks with separate local contexts, corresponding to the individual cases of the original disjunction. The structure of the subsequent proof pattern follows the rule $\vdash A_1 \vee A_2 \vee A_3 \Longrightarrow (A_1 \Longrightarrow C) \Longrightarrow (A_2 \Longrightarrow C) \Longrightarrow (A_3 \Longrightarrow C) \Longrightarrow C$.

assume $A_1 \vee A_2 \vee A_3$
moreover {
  assume A_1
  hence C ⟨*proof*⟩
}
moreover {
  assume A_2
  hence C ⟨*proof*⟩
}
moreover {
  assume A_3
  hence C ⟨*proof*⟩
}
ultimately have C **by** *blast*

These examples already illustrate the most basic use of "big-step" reasoning via degenerate calculations used together with automated proof tools. Here plain natural deduction is essentially scaled up to a larger fragment of first-order logic, somewhat depending on the capabilities of the underlying proof procedure.

Such techniques of synthesizing results in a forward manner may occasionally get used with some advantage in order to streamline Isar proof texts. Deeply nested backward patterns may especially benefit, as illustrated by the subsequent versions of iterated $\forall$ introduction.

have $\forall x\ y\ z.\ P\ x\ y\ z$
proof
  fix x
  show $\forall y\ z.\ P\ x\ y\ z$
  proof
    fix y
    show $\forall z.\ P\ x\ y\ z$
    proof
      fix z
      show $P\ x\ y\ z$ ⟨*proof*⟩
    qed
  qed
qed

{
  fix $x\ y\ z$
  have $P\ x\ y\ z$ ⟨*proof*⟩
}
then have $\forall x\ y\ z.\ P\ x\ y\ z$ **by** *blast*

Here the "calculation" happens to consist of a single proof block only, with the result being fed into the *blast* step via plain forward chaining of "**then have**", instead of the previous "**ultimately have**". In fact, "**ultimately**" happens to be equivalent to "**moreover note** *calculation* **then**" (cf. §3.3.3 and §6.2.2).

Thus we may understand the present technique of degenerate calculations as a natural generalization of Isar's **then** primitive (cf. §3.2.1): plain **then** enables the proof writer to continue with the most recent result immediately, avoiding explicit naming of previous facts; **moreover**/**ultimately** achieve a similar effect for an arbitrary number of results accumulated from several chunks of proof text.

More complex nesting of (mixed) logical connectives may be used in big-step reasoning as well. The only limits are those imposed by the capabilities of the ultimate proof method. Here the actual Isar proof processor merely takes care of the general flow of facts, including handling of nested proof contexts. Proper integration of individual results is left to the accidental virtues of the automated proof procedures that happen to be available.

The following example essentially covers simultaneous elimination of $\exists$, $\vee$, $\wedge$. This is still a very simple task for the *blast* method of Isabelle (see §7.3).

assume $A \vee (\exists x.\ P\ x) \vee (\exists y\ z.\ Q\ y\ z \wedge R\ z)$
moreover {
  assume A
  hence C ⟨*proof*⟩
}
moreover {
  fix x
  assume $P\ x$

```
  hence C ⟨proof⟩
}
moreover {
  fix y z
  assume Q y z and R z
  hence C ⟨proof⟩
}
ultimately have C by blast
```

In reality, one might even attempt to omit the explicit "covering" statement $A \lor (\exists x.\ P\ x) \lor (\exists y\ z.\ Q\ y\ z \land R\ z)$, and let the automated prover take care of this formal detail itself. Further contributing facts may be included as a separate **moreover** step, which is better put at the end of the calculation. Thus we happen to achieve a succinct presentation of "generalized case-splitting" patterns already discussed in §5.5.3.

```
{
  assume A
  hence C ⟨proof⟩
}
moreover {
  fix x
  assume P x
  hence C ⟨proof⟩
}
moreover {
  fix y z
  assume Q y z and R z
  hence C ⟨proof⟩
}
moreover note auxiliary-stuff
ultimately have C by blast
```

Recall that case-splitting with a single branch coincides with the simpler form of "generalized elimination" already covered by the **obtain** command (cf. §5.3). Such elimination patterns may certainly be rephrased in big-step reasoning style as well, although this turns out as slightly more awkward in practice.

```
assume ∃x y z. P x ∧ Q y z ∧ R z
then obtain x y z where P x and Q y z and R z by blast

assume ∃x y z. P x ∧ Q y z ∧ R z
moreover {
  fix x y z
  assume P x and Q y z and R z
  hence C ⟨proof⟩
}
ultimately have C by blast
```

Here the issue of avoiding explicit elimination statements (given as initial assumptions above) is the same as before, cf. the related discussion in §5.3.3. Note that the second version does require the explicit conclusion C, unlike **obtain** which has been designed to be independent of ultimate goal statements in the first place (§5.3).

Problems of big-step reasoning

Concerning the general issue of high-level proof processing, the previous examples suggest that we could in principle ignore most of the existing Isar infrastructure that is intended for fine-grained natural deduction proof composition (chapter 3), and be content with only a few basic elements to declare the overall proof structure, together with a sufficiently powerful proof procedure to fill in the remaining gaps. For example, one might be tempted to restrict the Isar language to **fix**, **assume**, **have**, **moreover**, **ultimately**, "**{**", "**}**", and "**by** *blast*".

Such an extreme view of big-step reasoning has been occasionally proposed as a natural approach to structured proof processing, especially by those with a special interest in automated reasoning.

[Dahn and Wolf, 1994] introduce a separate calculus for structured proofs in classical first-order logic, with separate means to achieve local results by virtue of automated reasoning. These concepts have been integrated into the ILF system [Dahn *et al.*, 1997], which has been able to process an adapted version of the full collection of Mizar articles [Mizar library] by heavy-duty first-order automation. Thus existing formalized proofs have been replayed successfully.

On the other hand, it is unclear how that system would perform in development of new applications. The original Mizar system [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] is particularly good at providing fine-grained error messages of failed inference steps, which are indispensable as user feedback for failed proof attempts. Existing automated reasoning tools are usually designed with different goals in mind, posing a serious problem to error recovery. For example, a "resolution prover" would transform the initial problem into an unintelligible internal normal form first, so any immediate feedback would be given at a level that is hard to follow by most users.

Harrison's "Mizar mode for HOL" [Harrison, 1996b] and Zammit's SPL [Zammit, 1999a] [Zammit, 1999b] show some tendency towards heavy automated proof tools as a key to high-level proof texts as well. Substantial parts of that work covers the issue of providing suitable automated reasoning for the underlying implementations of HOL. Nevertheless, both of these systems still provide a useful collection of basic proof steps based on simple inferences.

DECLARE [Syme, 1997a] [Syme, 1998] [Syme, 1999] diminishes the domination of automated reasoning a bit further. DECLARE still has a heavy combination of automated tools hard-wired (analogous to *auto* in Isabelle/HOL, see §7.3), but this is only one of its three basic principles of declarative theorem proving.

Speaking in Isar terms, the other two essentially correspond to advanced elimination and case-splitting principles (§5.3 and §5.5.3), and specific support for induction patterns (§5.4).

We argue that purely "automatic" checking of high-level proof texts is quite unsuitable for realistic applications. From the Isar perspective, automated reasoning needs to be restricted to its proper place of solving occasional proof obligations occurring terminally. Backed by the experience of Isar applications so far, we see this restrictive policy towards proof automation as a key factor to achieve a viable system supporting a broad range of formal developments.

Automated reasoning essentially suffers from two main deficiencies.

1. Failed proof search usually does not provide feedback for users.
2. Successful proof search is generally non-compositional.

The issue of failing gracefully is particularly important for development (and maintenance) of proofs. During a typical interactive session, invocations of automated tools fail over and over again, until the user is able to isolate all the bugs in statements and omissions of auxiliary facts eventually. Thus successful automated proof search actually turns out as the exception, rather than the rule. So instead of an automated proving, users would first and foremost need proper support for automated "*disproving*". In other words some tools to produce counter-examples systematically, cf. related issues covered by the KIV system [Reif, 1992]. Similar experience with the proof development cycle has been reported in a non-trivial case-study conducted with DECLARE [Syme, 1997a] [Syme, 1998] [Syme, 1999].

Compositionality is probably the key to scalable applications whatsoever. It generally means that smaller entities may be composed to larger ones in a modular fashion, without breaking down previous work. In the context of theorem proving two specific aspects need to be covered here in particular: *monotonicity* wrt. proof contexts and *invariance* wrt. instantiation of statements.

In terms of Isar, monotonicity means that additional elements of **fix**, **assume**, **have**, **note** etc. may be safely added to an existing proof body. It is easy to see that the basic Isar proof processing mechanism (§3.2.3) indeed fulfills this property, essentially due to monotonicity of the underlying higher-order backchaining rule (§2.4); the same holds for robustness against instantiation.

The situation changes immediately when automated proof methods enter the scene. In practical Isar proof development partial proofs routinely break down due to ill-behaved situations involving advanced methods like *blast*, *auto*, *force* (see §7.3). More powerful procedures typically cause more problems in practice than simpler ones (like *simp*). Consequently, a general "strength reduction" of the deductive tools that are sufficient to conduct meaningful Isar applications turns out as an important contribution to practical usability of the system.

As will be illustrated by concrete applications later on (chapter 8, chapter 9, chapter 10), Isar proofs largely refer to relatively simple deductive means, such as single-step inferences (using the fundamental methods *rule* or *this*, cf. §3.3.2) or plain higher-order rewriting (using *simp*, see §7.3).

Thus our common Isar proof style is able to achieve relatively robust proof outlines that may be analyzed in reasonable fine-grained steps, which may be checked interactively by the user. Typically there are only very few "hot spots" of terminal proofs by heavy automated methods (like "**by** *blast*") that need to be taken care of separately. The latter tend to demand substantial amount of time in practice, both from the user and the machine (cf. the related discussion in §10.7.2). So it is important to minimize these incidents of automated reasoning in sizable Isar developments.

In contrast, the present style of big-step reasoning emphasizes the very role of automated means: results gathered over large portions of proof text (probably with complex structured local contexts) need to be handled ultimately in a single big step. Note that the resulting problems are similar to those of "big-bang integration" of software components: a number of individual pieces finished beforehand are suddenly joined to a more complex system. Lacking any previous considerations of how individual elements may need to correlate, any failures of integration result in a single big problem to be treated in an ad-hoc fashion.

For example, reconsider one of the previous generalized case-splitting patterns, with just a few minor modifications in the individual proof blocks (maybe due to typographical errors, or some misunderstanding of the writer).

assume $A \lor (\exists\, x.\ P\ x) \lor (\exists\, y\ z.\ Q\ y\ z \land R\ z)$
moreover {
  assume A' **and** B
    — modified assumption A', additional one B
  hence C ⟨*proof*⟩
}
moreover {
  fix x
  assume $P\ x$
  hence C' ⟨*proof*⟩
    — modified conclusion C'
}
moreover {
  fix $y\ z\ w$
  assume $Q\ y\ z$ **and** $R'\ z\ w$
    — modified assumption $R'\ z\ w$, additional parameter w
  hence C ⟨*proof*⟩
}
ultimately have C
  ⋮

Here the individual sub-problems may still work accidentally; their respective proofs may not affect the overall situation anyway, because of compositional proof processing in Isar. After having successfully processed a large body of text, the user is ultimately faced with a failure to compose these pieces successfully by virtue of automated reasoning. Unfortunately, the latter would usually provide no clue what exactly went wrong, e.g. in the particular mismatch the individual local proof contexts against the initial covering statement (which may have been left implicit in the first place). Note that some modifications of the previous text may actually be valid transformations of the overall proof problem, independently if this has been intended by the writer.

Ontic [McAllester, 1988] essentially follows the general idea of big-step reasoning very faithfully. Its main paradigm is that of "socratic proofs", i.e. nested lemmas that are linked implicitly by a specific inference mechanism. Interestingly, McAllester rejects arbitrary "heuristic" procedures here as well, but focuses on "algorithmic" principles [McAllester, 1990]. Note that degenerate calculations in Isar could easily emulate Ontic proof schemes, essentially by replacing the *blast* method encountered above by an implementation of McAllester's inference mechanism.

A similar tendency to reject arbitrary proof search may be observed in Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999], where the builtin notion of "obvious inferences" [Rudnicki, 1987] has been deliberately limited to a fragment of classical first-order logic that may be decided quickly. This decision may be even one of the deeper reasons why Mizar users have been able to compile a large body of applications [Mizar library] over the years.

We see that the useful concept of degenerate calculations in Isar needs to be applied with some care, lest the resulting proof development result in serious inconveniences for both the writer and anybody doing maintenance work later on. The general liberality principle of Isar (cf. §1.3) does not stop users from abuse, though. It is left to the discernment of the proof writer to express advanced high-level proof schemes adequately.

Part III

Applications

Chapter 7

The Isabelle/HOL application environment

We outline the Isabelle/HOL application environment, as required for several "realistic" Isabelle/Isar examples covered later on. Apart from basic logical preliminaries of HOL, which are not of primary interest here, we review practically relevant issues of derived definitional mechanisms, advanced proof methods, and the main Isabelle/HOL library.

The Isar framework is essentially as generic as Isabelle itself, so different object-logics could be used for applications as well. On the other hand, practical work demands a sufficiently rich environment of tools and theories to be readily available. We commit ourselves to the existing Isabelle/HOL system in order to get started with only minimal Isar specific additions required.

7.1 The HOL logic

HOL [Gordon and Melham, 1993] [Gordon, 2000] has emerged from an old-fashioned system by Church, originally called the "Simple Theory of Types" [Church, 1940]. Viewed from the perspective of pure logic, several aspects of HOL might appear slightly peculiar at first sight, like the type definition primitive for example (see §7.1.2 and §8.6). On the other hand, the system has proven as a robust basis for viable theorem proving environments that take soundness issues seriously.

Despite being simplistic in several respects, HOL is capable of many sophisticated constructions inside the basic system, without requiring to change its very foundations. Occasionally, advanced concepts are possible *because* of seeming weaknesses of HOL, e.g. overloading [Wenzel, 1997] or extensible records [Naraschewski and Wenzel, 1998].

The Isabelle/HOL formulation [Nipkow *et al.*, 2001] is faithful to the original HOL family [Gordon and Melham, 1993], although the influence of the Isabelle system philosophy makes a difference in many details. In particular, special care has been taken to make the user-experience of HOL closer to traditional mathematics, e.g. by providing a separate type of sets apart from primitive predicates, together with common set-theory notation. Thus internal features of HOL may be largely hidden from the sight of casual readers of Isabelle/HOL theory developments.

The resulting system of simply-typed set theory is quite convenient to use in realistic applications. Compared to actual (untyped) set-theory (e.g. Isabelle/ZF [Paulson, 1993] [Paulson, 1995]), HOL avoids a number of complications such as the non-uniform treatment of propositions versus individuals in first-order logic, or typing issues that would need to be handled explicitly via set-membership reasoning instead of static typing.

Subsequently, we point out a few aspects of basic HOL concepts, as relevant to end-user applications. Further foundational details are discussed within the Isabelle/Isar example of chapter 8.

7.1.1 Simply-typed set theory

Isabelle/HOL may be best understood as as *simply-typed classical set-theory*. The general look-and-feel is close to the original formulation of set-theory according to Cantor, where sets and classes are treated uniformly (soundness of HOL is guaranteed by simple types [Church, 1940]). In particular, set-comprehension may be used naively, similar to common practice in informal mathematics. Types may usually be omitted from specifications due to automatic type-reconstruction, similar to ML or Haskell.

Isabelle/HOL provides the following basic notation for forming sets from predicates and general functions.

$\{x.\ P\ x\}$	set comprehension
$\{f\ x \mid x.\ P\ x\}$	mapped set comprehension
$f\ `\ A$	set image

Further set-theory concepts are written as usual, e.g. membership $x \in A$, union $A \cup B$, intersection $A \cap B$, difference $A - B$, "big" union $\bigcup x.\ B\ x$ (over the type of x), or $\bigcup x \in A.\ B\ x$ (over the set A). By coincidence, most of the basic notation of Isabelle/HOL closely follows that of our informal background language outlined in §2.1.

The pure "logic" of HOL coincides where set-theory notions, with boolean expressions and predicates replacing sets. Logical notation is fairly standard, including quantification $\forall x.\ P\ x$ and $\exists x.\ P\ x$, set-bounded quantification $\forall x \in A.\ P\ x$ and $\exists x \in A.\ P\ x$, and the standard connectives of $\wedge$, $\vee$, $\longrightarrow$ etc.

Note that this internal object-logic of HOL is mostly used for building up complex expressions involving boolean values. The primary rules of reasoning are typically formulated at the Isabelle meta-level, using $\bigwedge/\Longrightarrow$ connectives instead of HOL's own collection of $\forall$, $\exists$, $\longrightarrow$, $\wedge$, $\vee$, $\neg$ etc. The general discipline of preferring meta-level rules over object-formulae considerably reduces the formal noise in actual proofs, since the outer structure of logical statements need not be decomposed explicitly in proofs. The basic mechanism of higher-order backchaining of the Isabelle/Pure framework (§2.4) directly operates on such rules as expected. The Isar proof language (chapter 3) provides immediate links to the very same principles.

Atomic meta-level propositions require separate concrete syntax of *PROP A* (where *A* is a term of type *prop*); omitting the *PROP* marker would make the proposition "*A*" an object-logic judgment, where *A* is a boolean expression. Note that the latter involves a hidden coercion from *bool* to *prop*, which is traditionally called *Trueprop* in Isabelle.

Functions play an important role in HOL. The Isabelle version uses mostly standard notation from λ-lambda calculus, and modern higher-order programming languages like Haskell or ML (e.g. see [Paulson, 1991]).

$f\ a$	application
$\lambda x.\ b[x]$	abstraction
$let\ x = a\ in\ b[x]$	local binding
$if\ A\ then\ a\ else\ b$	conditional expression
$case\ p\ x \Rightarrow b[x] \mid \ldots$	simple pattern matching
$f\ (x := y)$	point-wise functional update

Despite this notational coincidence, HOL does *not* quite resemble programming languages. In particular, there is no artificial restriction to computable functions, although this could be easily formalized within HOL, e.g. [Regensburger, 1995] [Müller *et al.*, 1999].

HOL functions are inherently total; partial ones may be easily represented using the polymorphic *option* datatype, which consists of *None* or *Some x* elements. The following basic operations are available on the common type $'a \Rightarrow 'b\ option$, which may be considered as a the canonical model for partial mappings.

$empty \equiv \lambda x.\ None$	empty mapping
$e\ (x \mapsto y) \equiv e\ (x := Some\ y)$	assignment of defined values
$dom\ e \equiv \{x.\ e\ x \neq None\}$	set of defined values (domain)

7.1.2 Primitive definitions

The HOL logic is based on a tiny axiomatic kernel [Gordon and Melham, 1993] stating only the most fundamental facts of classical set-theory within a simply-typed setting. According to established tradition of HOL methodology, any extensions of existing theories have to be *definitional*: only certain disciplined axiom schemes may be given. Thus several important well-formedness properties

of theories are preserved, without requiring the user to reconsider the full meta-theory of HOL over and over again.

Formal objects of HOL are differentiated into (simple) types and λ-terms, although both may be interpreted as sets in the standard model theory [Pitts, 1993]. Consequently, there are two separate definitional primitives, namely *constant definitions* and *type definitions*. These mechanisms turn out as quite dissimilar wrt. the technical specification and meta-theoretical properties.

Isabelle/HOL only provides a separate primitive for type definitions [Nipkow *et al.*, 2001] [Wenzel, 2001a]. Constant definitions are inherited from Isabelle/Pure [Paulson, 2001a].

Constant definitions

The **constdefs** element of Isabelle/Pure provides a user-level interface to basic constant definitions, including conditional equations (cf. §2.3). The specification consists of any number of pairs of constant declaration and definitional equality; the latter has to cover the same most-general type scheme as given in the declaration (this restriction prevents unintended overloading).

constdefs
  $a :: nat$
  $a \equiv 1$
  $b :: nat \Rightarrow nat$
  $a \leq n \Longrightarrow b\ n \equiv n - a$

The result of **constdefs** is a set of theorems, corresponding to the original equations; each theorem is named after its constant, e.g. *a-def* and *b-def*.

The separate **consts** and **defs** commands provide direct access to the underlying primitives of constant declarations and definitions. Thus definitions may be given independently of declarations. Several advanced definitional packages require constants to be declared separately, e.g. **inductive** (see §7.2.1) and **primrec** (see §7.2.2). Furthermore, **defs** allows multiple definitions to be given on non-overlapping type schemes by virtue of overloading (cf. §2.3).

consts
  $c :: 'a$
defs (**overloaded**)
  c-nat-def: $c :: nat \equiv 0$
  c-bool-def: $c :: bool \equiv False$
  c-prod-def: $c :: 'a \times 'b \equiv (c :: 'a,\ c :: 'b)$

An overloaded definition like this entails obvious results about more complex type instances, essentially via primitive recursion over the syntactic structure of type expressions.

lemma $c :: (nat \times bool) \times 'a \equiv ((0,\ False),\ c :: 'a)$
  by (*simp only*: *c-nat-def c-bool-def c-prod-def*)

Nothing specific may be derived about unspecified types yet, like $c :: 'a \Rightarrow 'b$. The collection of definitional equations may be augmented incrementally later on, e.g. as follows.

defs (**overloaded**)
  c-fun-def: $c :: 'a \Rightarrow 'b \equiv \lambda x :: 'a.\ c :: 'b$

It is important to note that overloading inherently introduces underspecification in the theory, since it is impossible to cover all possible HOL type schemes uniformly. Underspecification prevents a number of meta-theoretical properties of Isabelle/HOL (e.g. preservation of completeness or decidability of certain sub-theories), but it turns out as very useful in advanced applications like axiomatic type classes [Wenzel, 1997] or object-oriented verification [Naraschewski and Wenzel, 1998]. In fact, the HOL logic does not admit very strong meta-theoretical properties in the first place, even in its original formulation [Gordon and Melham, 1993] [Pitts, 1993] without overloading.

Type definitions

The **typedef** command of Isabelle/HOL provides a convenient interface to the specific HOL notion of type definitions (see also §8.6 for foundational issues). The basic idea is to abstract a non-empty subset of an existing type expression into a new type. The Isabelle/Isar version admits the required non-emptiness proof to be performed immediately in the text [Wenzel, 2001a], as indicated in the example below.

typedef *even* = $\{n :: nat.\ \exists k.\ n = 2 * k\}$
proof
  show $0 \in$ *?even* **by** *simp*
qed

While the poof obligation of **typedef** is actually an abstract existential statement, it has been reduced in the initial **proof** step to demand a concrete witness (via $\exists$ introduction). The subsequent **show** statement takes care of this main aspect of the **typedef** proof. Note that the term abbreviation *?even* of the representing set is made available in the initial proof context automatically.

The result of the above **typedef** is a new type *even* according to the principle of HOL type definitions (see §8.6), as well as a constant constant definition of the same name for the representing set within the old type. Rules that characterize bijections (*Abs-even* and its inverse *Rep-even*) between the representing set and the new type are provided as well.

The internal bijection axioms (see §8.6.1) are slightly too primitive for direct use. Isabelle/Isar provides alternative formulations that are suitable for high-level reasoning. In particular, the surjection part will be expressed as rules for cases and induction in order to support canonical representation proofs (cf. §5.3 and §5.4). This is illustrated for *even* by the subsequent proof patterns.

```
obtain b where a = Abs-even b and b ∈ even
  by (cases a)

fix b assume b ∈ even
then obtain a where b = Rep-even a
  by cases

fix x :: even
have P x
proof (induct x)
  fix y assume y ∈ even
  thus P (Abs-even y) ⟨proof⟩
qed

fix y assume y ∈ even
hence P y
proof induct
  fix x
  show P (Rep-even x) ⟨proof⟩
qed
```

The *cases* and *induct* versions are essentially equivalent, due the degenerate representation (without recursion) involved here. Nevertheless, both formats have their use in actual applications. Cases are usually better suited to represent concrete expressions of the new type or the representing set, respectively. Induction is more appropriate to establish universal properties.

7.2 Advanced definitional packages

In principle, the primitive definitional mechanisms for constant and type definitions given in §7.1.2 are already sufficient to build up substantial applications on top of basic HOL, while proceeding in a disciplined manner without ad-hoc axiomatization of new concepts. In reality, further advanced definitional concepts are required to support sizable developments.

Several derived definitional mechanisms have emerged in the HOL tradition over many years [Gordon, 2000], most notably those of inductive sets and types together with recursive functions; see also the outline given in [Berghofer and Wenzel, 1999]. It is important to note that such high-level concepts are usually built on top of existing primitives; the meta-theory of HOL need not be reconsidered by those who develop new packages. Implementors do not even need to understand the HOL logic in every detail, some rough idea of typed set-theory will be sufficient for most purposes.

See figure 7.1 for the collection of advanced definitional packages provided by Isabelle/HOL. Here **constdefs** and **typedef** are taken as primitives (cf. §7.1.2).

The others will be briefly reviewed below, with some focus on specific details that are particularly relevant to Isar.

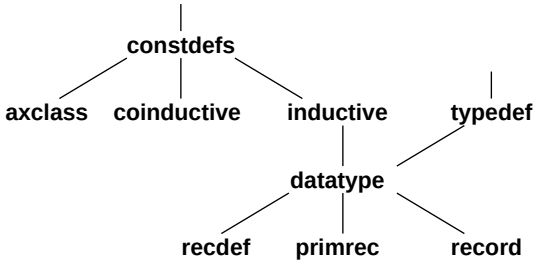

Figure 7.1: Definitional packages of Isabelle/HOL

7.2.1 Inductive sets and types

The most fundamental advanced definitional mechanisms of Isabelle/HOL are: **inductive** (and **coinductive**) for sets defined via Knaster-Tarski fixed-points [Paulson, 1994], and **datatype** for arbitrarily branching tree types (optionally with mutual or indirect recursion). See [Berghofer and Wenzel, 1999] for a more detailed exposition of various classes of **datatype** definitions in HOL. Note that there is presently no package for co-datatypes, even co-inductive *sets* are rarely used in existing Isabelle/HOL applications.

Inductive sets

The **inductive** package takes a collection of introduction rules specified in the theory text; internally it produces a number of basic definitions for least-fixed points on the complete lattice of power sets (e.g. see [Davey and Priestley, 1990]); it then automatically proves the introduction rules as specified in the text, together with canonical eliminations (resulting from the fixed-point equation), and the induction rule (from the least fixed-point property). Eliminations are declared as standard rules for the *cases* method, likewise is induction for the *induct* method (cf. §5.4).

The original version of **inductive** [Paulson, 1994] has been improved to allow nesting of standard logical connectives (including quantifiers), where the canonical polarity rules are handled automatically in the internal monotonicity proof [Berghofer and Wenzel, 1999]. Most importantly, the meta-logical connectives $\Longrightarrow/\bigwedge$ may be involved as well, allowing arbitrarily nested rules to be given as introductions [Wenzel, 2001a]. In particular, *infinitely branching* inductive definitions may be expressed quite naturally, without recourse to separate connectives of the object-logic.

To illustrate the latter technique we present a definition of the σ-algebra generated by a given collection of basic sets (this is a standard mathematical concept of measure theory). The most interesting case is that of *Union* given below, where the inductive parameter is indexed over type *nat*.

consts
  sigma-algebra :: *'a set set* ⇒ *'a set set*
inductive *sigma-algebra A*
  intros
    basic: $a \in A \Longrightarrow a \in$ *sigma-algebra A*
    UNIV: *UNIV* ∈ *sigma-algebra A*
    complement: $a \in$ *sigma-algebra A* $\Longrightarrow -a \in$ *sigma-algebra A*
    Union: $(\bigwedge i{::}nat.\ a\ i \in$ *sigma-algebra A*$) \Longrightarrow (\bigcup i.\ a\ i) \in$ *sigma-algebra A*

The following proof illustrates how these introductions may be used as ordinary inference rules, while we establish the dual of *Union* via the de-Morgan property of sets.

theorem *Inter*: $(\bigwedge i{::}nat.\ a\ i \in$ *sigma-algebra A*$) \Longrightarrow (\bigcap i.\ a\ i) \in$ *sigma-algebra A*
proof −
  assume $\bigwedge i{::}nat.\ a\ i \in$ *sigma-algebra A*
  hence $\bigwedge i{::}nat.\ -(a\ i) \in$ *sigma-algebra A* **by** (*rule complement*)
  hence $(\bigcup i.\ -(a\ i)) \in$ *sigma-algebra A* **by** (*rule Union*)
  hence $-(\bigcup i.\ -(a\ i)) \in$ *sigma-algebra A* **by** (*rule complement*)
  also have $-(\bigcup i.\ -(a\ i)) = (\bigcap i.\ a\ i)$ **by** *simp*
  finally show *?thesis* **.**
qed

Apparently, the proof works out in a straight-forward manner by a number of meaningful steps, without requiring any unexpected formal twists. In contrast, the subsequent version uses alternative formulations of *Union* and *Inter*, where the infinite branching over natural numbers is expressed within the object-logic via separate ∀ (according to existing practice of Isabelle/HOL).

lemma *Union′*: $\forall i{::}nat.\ a\ i \in$ *sigma-algebra A* $\Longrightarrow (\bigcup i.\ a\ i) \in$ *sigma-algebra A*
  ⟨*proof*⟩

lemma *Inter′*: $\forall i{::}nat.\ a\ i \in$ *sigma-algebra A* $\Longrightarrow (\bigcap i.\ a\ i) \in$ *sigma-algebra A*
proof −
  assume $\forall i{::}nat.\ a\ i \in$ *sigma-algebra A*
  hence $\bigwedge i{::}nat.\ a\ i \in$ *sigma-algebra A* **..** — formal noise
  hence $\bigwedge i{::}nat.\ -(a\ i) \in$ *sigma-algebra A* **by** (*rule complement*)
  hence $\forall i{::}nat.\ -(a\ i) \in$ *sigma-algebra A* **..** — formal noise
  hence $(\bigcup i.\ -(a\ i)) \in$ *sigma-algebra A* **by** (*rule Union′*)
  hence $-(\bigcup i.\ -(a\ i)) \in$ *sigma-algebra A* **by** (*rule complement*)
  also have $-(\bigcup i.\ -(a\ i)) = (\bigcap i.\ a\ i)$ **by** *simp*
  finally show *?thesis* **.**
qed

We see that introduction and elimination of $\forall$ has to be taken care of explicitly. This was not required in our original version, since basic inferences of the pure framework already handle local $\Longrightarrow/\bigwedge$ contexts as expected (§2.4).

Strictly speaking, the proof above still relies on uniform treatment of non-atomic statements within Isar proof texts (cf. §5.2.5). Suppressing this as well, we would get an even more cumbersome proof of *Inter′* as follows.

lemma *Inter′*: $\forall i{::}nat.\ a\ i \in$ *sigma-algebra* $A \Longrightarrow (\bigcap i.\ a\ i) \in$ *sigma-algebra* A
proof −
  assume *a*: $\forall i{::}nat.\ a\ i \in$ *sigma-algebra* A
  have $\forall i{::}nat.\ -(a\ i) \in$ *sigma-algebra* A
  proof
    fix *i* **from** *a* **have** $a\ i \in$ *sigma-algebra* A **..**
    thus $-(a\ i) \in$ *sigma-algebra* A **by** (*rule complement*)
  qed
  hence $(\bigcup i.\ -(a\ i)) \in$ *sigma-algebra* A **by** (*rule Union′*)
  hence $-(\bigcup i.\ -(a\ i)) \in$ *sigma-algebra* A **by** (*rule complement*)
  also have $-(\bigcup i.\ -(a\ i)) = (\bigcap i.\ a\ i)$ **by** *simp*
  finally show *?thesis* **.**
qed

Here all statements in the text (except the main one) have become atomic propositions. While this does not pose any fundamental limitation "in principle", it makes a big difference for general usability of the system in practice.

On the other hand, the additional formal twists due to object-level connectives exposed in this example would not make a big difference for traditional Isabelle proof scripts, as these usually contain a large number of technical clarification steps anyway. Unstructured proof scripts tend to contain lots of formal noise unnoticed.

We shall also take a brief look at the induction rule emerging from the above **inductive** definition:

$x \in$ *sigma-algebra* $A \Longrightarrow$
$(\bigwedge a.\ a \in A \Longrightarrow P\ a) \Longrightarrow$
P *UNIV* $\Longrightarrow$
$(\bigwedge a.\ a \in$ *sigma-algebra* $A \Longrightarrow P\ a \Longrightarrow P\ (-\ a)) \Longrightarrow$
$(\bigwedge a.\ (\bigwedge i.\ a\ i \in$ *sigma-algebra* $A) \Longrightarrow (\bigwedge i.\ P\ (a\ i)) \Longrightarrow P\ (\bigcup i.\ a\ i)) \Longrightarrow P\ x$

Naturally, the nesting of meta-level connectives given in the specification of introductions carries over to induction as well. Users of traditional Isabelle proof scripts would normally abhor complex meta-level statements of this kind, due to the inherent limitation of basic tactics to rules that consist of atomic statements only [Paulson and Nipkow, 1994]. As already pointed out before, it is easy to cope with this situation in proper Isar proof texts, since there is nothing special about non-atomic propositions (§5.2.5); the *induct* method takes special care to preserve this uniform view on non-atomic statements (§5.4.5).

The induction rule for *sigma-algebra A* is already declared for use in standard elimination patterns of induction (§5.4). Thus we may perform "rule induction" wrt. the definition of *sigma-algebra A* as follows.

assume $x \in$ *sigma-algebra A*
hence $P\ x$
proof *induct*
  fix a **assume** $a \in A$
  thus $P\ a$ ⟨*proof*⟩
next
  show P *UNIV* ⟨*proof*⟩
next
  fix a **assume** $a \in$ *sigma-algebra A* **and** $P\ a$
  thus $P\ (-a)$ ⟨*proof*⟩
next
  fix a **assume** $\bigwedge i{::}nat.\ a\ i \in$ *sigma-algebra A* **and** $\bigwedge i.\ P\ (a\ i)$
  thus $P\ (\bigcup i.\ a\ i)$ ⟨*proof*⟩
qed

As the inductive assumptions and side-conditions directly reflect the textual complexity of **inductive** definition, we may consider to use the infrastructure of symbolic case names offered by the *induct* method (cf. §5.4).

assume $x \in$ *sigma-algebra A*
hence $P\ x$
proof (*induct* (*open*) $P\ x$)
  case *basic*
  thus $P\ a$ ⟨*proof*⟩
next
  case *UNIV*
  thus P *UNIV* ⟨*proof*⟩
next
  case *complement*
  thus $P\ (-a)$ ⟨*proof*⟩
next
  case *Union*
  thus $P\ (\bigcup i.\ a\ i)$ ⟨*proof*⟩
qed

This compact induction pattern requires additional instantiations of P and x given in advance. Furthermore, we have also declared the scope of local parameters to be "(*open*)", which makes the terminology of local parameters from the original **inductive** definition appear implicitly in the proof text (cf. §5.4.4).

Inductive types

The **datatype** package of Isabelle/HOL provides a convenient interface for a very general class of tree structures represented in classical set-theory. Only

the collection of constructors (with types) has to be specified by the user. The internal construction is based on **inductive** definitions over a set-theoretic universe that has been defined within HOL beforehand [Paulson, 1994] [Berghofer and Wenzel, 1999], the result is then abstracted via the **typedef** primitive (cf. §7.1.2). Furthermore, a large number of additional infrastructure is derived automatically behind the scenes, including induction rules and support for recursion (both primitive and general well-founded one, see also §7.2.2).

In practice, the most important datatype is that of lists over some existing type. The subsequent definition of *'a list* has been taken from the main library of Isabelle/HOL [Nipkow *et al.*, 2001].

datatype *'a list* =
  Nil ([])
| *Cons 'a 'a list* (**infixr** # 65)

The induction provided by this definition has already been declared for use in plain introduction patterns of the *induct* method (cf. §5.4). Thus we may perform standard structural induction as follows.

have *P xs*
proof (*induct xs*)
  show *P* [] ⟨*proof*⟩
next
  fix *x xs* **assume** *P xs*
  thus *P* (*x* # *xs*) ⟨*proof*⟩
qed

Since **datatype** supports arbitrary branching (over any existing HOL type), the issues raised in the previous discussion of complex meta-level rules arising from **inductive** definition essentially apply here as well, albeit to a lesser degree. The basic inductive structure underlying **datatype** definitions is much simpler than common **inductive** ones.

Consider the subsequent example of nested partial functions as one of the more complex cases encountered in practice so far (see the theory *Nested-Environment* in [Bauer *et al.*, 2001], and the application in chapter 10).

datatype (*'a*, *'b*, *'c*) *env* =
  Val 'a
| *Env 'b 'c* ⇒ (*'a*, *'b*, *'c*) *env option*

The main induction rule of type *env* involves a nested meta-level ⋀-quantifier corresponding the indirect recursion via a function type in the second case.

(⋀*a*. *P*₁ (*Val a*)) ⟹
(⋀*b fun*. (⋀*x*. *P*₂ (*fun x*)) ⟹ *P*₁ (*Env b fun*)) ⟹
*P*₂ *None* ⟹ (⋀*env*. *P*₁ *env* ⟹ *P*₂ (*Some env*)) ⟹ *P*₁ *env*

Such higher rules work very well in Isar proof texts, see also the full theory of nested environments in [Bauer *et al.*, 2001]. Certainly, traditional Isabelle

tactic scripts would quickly run into serious inconveniences due to the negative nesting of meta-level connectives.

7.2.2 Recursive function definitions

Primitive recursion

The **datatype** package (cf. §7.2.1) provides standard combinators for structural primitive recursion. In fact, primitive recursion covers a very large range of function definitions within the higher-order framework of HOL. Also note that there is no inherent restriction to computable functions involved here.

The **primrec** package offers a simple user-interface for this definitional mechanism, requiring only the intended equations to be given by the user; these are used to determine the primitive definition inside, and are returned as proven theorems [Nipkow *et al.*, 2001]. See also [Berghofer and Wenzel, 1999] for a more extensive discussion of both **datatype** and **primrec**, including some further issues of seamless integration of several definitional packages.

As a simple example of **primrec** we define the append function over $'a\ list$ as introduced before (cf. §7.2.1). Here the recursion operates over the first argument, where we need to give canonical constructor patterns of the datatype. The second argument is a fixed parameter of the recursive definition. This specification coincides with the official one of Isabelle/HOL [Nipkow *et al.*, 2001].

consts
  $append :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$ (**infixr** @ 65)
primrec
  $[] \mathbin{@} ys = ys$
  $(x \# xs) \mathbin{@} ys = x \# (xs \mathbin{@} ys)$

The **primrec** equations are available as a list of theorems called *append.simps*; these facts are also declared as standard simplification rules (see also §7.3). Consequently, the Simplifier essentially performs $\beta\iota$-reduction by default (speaking in terms of Coq [Barras *et al.*, 1999]).

Apart from simplification, the canonical technique to establish results about **primrec** functions is to use the generic datatype induction rule, which may be accessed via the standard introduction pattern of the *induct* method (cf. §5.4).

theorem *append-assoc*: $(xs \mathbin{@} ys) \mathbin{@} zs = xs \mathbin{@} (ys \mathbin{@} zs)$ (**is** $?P\ xs$)
proof (*induct xs*)
  show $?P\ []$ **by** *simp*
next
  fix $x\ xs$ **assume** $?P\ xs$
  thus $?P\ (x \# xs)$ **by** *simp*
qed

Alternatively, we may invoke the infrastructure of symbolic cases offered by the *induct* method, provided that a full instantiation is given in advance. Since

the default terminology of local parameters produced by the **datatype** package is somewhat cryptic (being derived from the names of types involved in the inductive structure) we would better use the renamed rule associated with the **primrec** definition itself.

```
theorem (xs @ ys) @ zs = xs @ (ys @ zs)  (is ?P xs)
proof (induct (open) ?P xs rule: append.induct)
  case Nil
  thus ?P [] by simp
next
  case Cons
  thus ?P (x # xs) by simp
qed
```

Apparently, the latter scheme is slightly more heavyweight than the previous version. In practice, this pattern of referring to implicit parameters should be mainly useful with more complex structures than plain lists encountered here.

General recursion

Apart from higher-order primitive recursion, Isabelle/HOL also provides general well-founded function definitions via Slind's **recdef** package (which is also known as "TFL") [Slind, 1996] [Slind, 1997]. The main advantages of **recdef** over **primrec** are the theoretically more general recursion scheme, and general pattern matching of constructor expressions given in the function arguments (including overlapping patterns with left-to-right precedence).

The user-interface of **recdef** is similar to **primrec**: given a number of equations, the package performs appropriate primitive definitions inside, and returns the result as proven theorems. The exact collection of resulting rules may vary from the original specification, though, depending on the unwinding of constructor patterns performed internally.

Due to the very general approach of **recdef**, its internal proof process turns out to be quite complicated in practice: **recdef** hardwires rather heavy automated proof tools that happens to work well for most simple cases, but are hard to control by users in general. The internal proof process consists of multiple stages, including well-foundedness of the determined recursion behavior of the function definition, as well as actual termination wrt. that relation. Following the tradition of fully automated reasoning here, these proof obligations need to be finished by standard tactics inside, which may only be controlled indirectly by additional hints to be given beforehand.

See also [Nipkow *et al.*, 2001] and [Nipkow and Paulson, 2001] for further explanations on how to make **recdef** work in practice. Note that the original version of the TFL package [Slind, 1996] [Slind, 1997] draws from the Gordon HOL tradition of system organization [Gordon and Melham, 1993] [Gordon, 2000], rather than the Isabelle one.

We argue that slightly better integration of **recdef** with the Isar proof language needs to be provided before the powerful definitional mechanism of TFL may get used more widely in Isabelle/HOL. Essentially, the proof process needs to be presented as an Isar goal statement (maybe with auxiliary proof context declarations to accommodate the different stages), similar to **typedef** (§7.1.2) or **instance** (see §7.2.4). The resulting infrastructure should enable users to finish simple recursive definitions by canonical automated proofs like "**by** *simp*" or "**by** *auto*", or be able to decompose complex ones into well-defined sub-problems. See also §7.5.1 for further discussion of the practically important issue of advanced specification mechanisms versus interactive proof.

7.2.3 Extensible records

Record types provide a high-level view on properly nested pairs, with separate operations for field selection and update [Naraschewski and Wenzel, 1998]. In Isabelle/HOL record types need to be declared explicitly via the **record** package as illustrated below.

record *foo* =
  $x :: nat$
  $y :: nat$
  $z :: nat$

This definition introduces type *foo* with notation $(\!|x :: nat, y :: nat, z :: nat|\!)$. Concrete record expressions may be written as $(\!|x = a, y = b, z = c|\!)$, the function $foo.make :: nat \Rightarrow nat \Rightarrow nat \Rightarrow foo$ yields the very same result.

Fundamental record operations include selection and update, both are named after the fields involved. For example, $x :: foo \Rightarrow nat$ and $x\text{-}update :: nat \Rightarrow foo \Rightarrow foo$; an expression $x\text{-}update\ a\ r$ may be written as $r\ (\!|x := a|\!)$. Nested updates $r\ (\!|x := a, y := b|\!)$ are available as well; here fields may be repeated and given in any order.

Record types and operations are polymorphic wrt. the "rest" of the nested field constructors involved. Due to the way that simple types (and type inference) work in HOL, this is already sufficient to provide a very useful notion of *extensible records* essentially for free [Naraschewski and Wenzel, 1998]. For example, we may add further fields to the existing record type *foo* as follows.

record *bar* = *foo* +
  $w :: bool$

Since *bar* is a proper type instance of *foo*, we may apply the previous operations without further ado. Both selectors and updates may be transferred to the extended record type, just by virtue of schematic polymorphism.

have $y\ (\!|x = a, y = b, z = c, w = d|\!) = b$
  by *simp*

have $(\!|x = a, y = b, z = c, w = d|\!)\ (\!|y := b'|\!) =$
$(\!|x = a, y = b', z = c, w = d|\!)$
by *simp*

The "rest" of a record type may be accessed directly as well, using the improper *more* field; concrete record notation uses "..." (three dots) here. For example, general schemes of *foo* are written as $(\!|x :: nat, y :: nat, z :: nat, \ldots :: 'a|\!)$ for types and $(\!|x = a, y = b, z = c, \ldots = rest|\!)$ for term expressions.

See also [Naraschewski and Wenzel, 1998] for further discussion of extensible record types in simply-typed HOL, including applications to hierarchies of mathematical structures, and object-oriented verification.

7.2.4 Axiomatic type classes

Apart from simple types and λ-terms already present in traditional HOL formulations, the Isabelle version provides a third syntactic layer of order-sorted *type classes* [Nipkow, 1993] [Nipkow and Prehofer, 1993]. First of all, type classes merely provide a separate qualification of HOL types, without any immediate logical impact yet. Based on a few simple observations of naive polymorphism in HOL, general classes (and relations) of types may be represented within the meta-logic. Thus one may give an immediate interpretation of the concepts of order-sorted type signatures as propositions of the pure logical framework [Wenzel, 1997].

The latter observation gives rise to separate specification elements of **axclass** and **instance**, for definition of "axiomatic" type classes and instantiation (with proof), respectively [Wenzel, 2001a]. With overloaded constant definitions (cf. §2.3), one arrives at a light-weight mechanism of abstract theories that is tightly integrated with the Isabelle type system [Wenzel, 1997] [Wenzel, 2001e].

We briefly review the canonical example of abstract algebraic structures, namely monoids and groups. First of all, we declare polymorphic operations, and define axiomatic type classes as predicates over these. The "argument" of a type class is the polymorphic type parameter involved here. Note that *term* refers to the universal (syntactic) class of Isabelle/HOL types.

consts
product :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\circ$ 70)
inverse :: $'a \Rightarrow 'a$ ((-$^{-1}$) [1000] 999)
unit :: $'a$ (1)

axclass *monoid* $\subseteq$ *term*
assoc: $(x \circ y) \circ z = x \circ (y \circ z)$
left-unit: $1 \circ x = x$
right-unit: $x \circ 1 = x$

axclass *group* $\subseteq$ *term*
  assoc: $(x \circ y) \circ z = x \circ (y \circ z)$
  left-unit: $1 \circ x = x$
  left-inverse: $x^{-1} \circ x = 1$

We may now derive abstract properties based on this axiomatization, such as the fact $x \circ x^{-1} = 1$ of general group theory.

theorem *group-right-inverse*: $(x::'a::group) \circ x^{-1} = 1$
proof −
  have $x \circ x^{-1} = 1 \circ (x \circ x^{-1})$
    by (*simp only*: *group.left-unit*)
  also have $\ldots = (1 \circ x) \circ x^{-1}$
    by (*simp only*: *group.assoc*)
  also have $\ldots = (((x^{-1})^{-1} \circ x^{-1}) \circ x) \circ x^{-1}$
    by (*simp only*: *group.left-inverse*)
  also have $\ldots = ((x^{-1})^{-1} \circ (x^{-1} \circ x)) \circ x^{-1}$
    by (*simp only*: *group.assoc*)
  also have $\ldots = ((x^{-1})^{-1} \circ 1) \circ x^{-1}$
    by (*simp only*: *group.left-inverse*)
  also have $\ldots = (x^{-1})^{-1} \circ (1 \circ x^{-1})$
    by (*simp only*: *group.assoc*)
  also have $\ldots = (x^{-1})^{-1} \circ x^{-1}$
    by (*simp only*: *group.left-unit*)
  also have $\ldots = 1$
    by (*simp only*: *group.left-inverse*)
  finally show *?thesis* **.**
qed

It is easy to see that groups are monoids as well, since the *right-unit* property may be derived as a theorem (using *group-right-inverse* from above). This inclusion may be formally reflected within the Isabelle type signature [Wenzel, 1997], including a separate **instance** proof as given below.

instance *group* $\subseteq$ *monoid*
proof
  fix $x\ y\ z :: 'a::group$
  show $(x \circ y) \circ z = x \circ (y \circ z)$ **by** (*rule group.assoc*)
  show $1 \circ x = x$ **by** (*rule group.left-unit*)
  show $x \circ 1 = x$
  proof −
    have $x \circ 1 = x \circ (x^{-1} \circ x)$
      by (*simp only*: *group.left-inverse*)
    also have $\ldots = (x \circ x^{-1}) \circ x$
      by (*simp only*: *group.assoc*)
    also have $\ldots = 1 \circ x$
      by (*simp only*: *group-right-inverse*)

```
    also have ... = x
      by (simp only: group.left-unit)
    finally show ?thesis .
  qed
qed
```

Concrete instantiations of axiomatic type classes may be given as well. Below we present the type of lists (over arbitrary argument types) as a monoid by defining $\circ$ as append and 1 as nil. Note that this instantiation technique relies on overloaded constant definitions in generic HOL [Wenzel, 1997] (cf. §2.3).

```
defs (overloaded)
  product-list-def: xs ∘ ys ≡ xs @ ys
  unit-list-def: 1 ≡ []

instance list :: (term) monoid
proof
  fix xs ys zs :: 'a list
  show (xs ∘ ys) ∘ zs = xs ∘ (ys ∘ zs)
    by (simp only: product-list-def append-assoc)
  show 1 ∘ xs = xs
    by (simp only: product-list-def unit-list-def append.simps)
  have xs @ [] = xs by (induct xs) simp-all
  thus xs ∘ 1 = xs
    by (simp only: product-list-def unit-list-def)
qed
```

Due to the very nature of overloaded definitions, instantiations for concrete structures may be only given once for each non-overlapping pattern of types [Wenzel, 1997] [Wenzel, 2001e]. Multiple views on the same HOL type would typically require isomorphic copies via **typedef** or **datatype**. Moreover, HOL type constructors may act like "functors" on type classes, as in the canonical example of direct binary products like $* :: (monoid, monoid)\ monoid$.

See also [Wenzel, 2001b] for a development of basic lattice theory that demonstrates further advanced techniques of axiomatic type classes.

7.3 Automated proof methods

The HOL logic turns out as a viable platform for a broad range of existing automated reasoning techniques. Generic Isabelle provides two main modules to build up powerful proof tools for object-logics: the "Simplifier" [Paulson and Nipkow, 1994] and the "Classical Reasoner" [Paulson, 1997]; the latter includes an advanced implementation based on existing tableau prover technology [Paulson, 1999]. Combinations of generic automated proof tools are available as well.

From this perspective, a low-level presentation of Isabelle/HOL via primitive axioms and derived rules (chapter 8) loses some significance to end-users, although certain inherent virtues of primitive HOL simplify the construction of proof tools considerably (e.g. the syntactic treatment of types [Lamport and Paulson, 1999]).

7.3.1 Incorporating arbitrary proof tools

The pure Isar framework is *independent* of any particular object-logic features and specific prover support. The key concepts of Isar proof processing (cf. §3.2.3) merely depend on the generic mechanisms of higher-order unification and back-chaining, which may be even understood as the most basic principles of (nested) natural deduction [Paulson, 1986] [Paulson, 1989] [Paulson, 1990].

Arbitrary proof tools may be incorporated into Isar via the interface of proof methods (cf. the interpretation $\mathcal{M}$ in §3.2.3). It is technically quite easy to turn existing Isabelle tactics into Isabelle/Isar methods [Wenzel, 2001a]. On the other hand, some care is required in order to achieve an appropriate high-level view on the large number of variant forms of tactics and tactic combinations that have emerged over time in Isabelle [Paulson, 2001b].

Isabelle/Isar follows a certain classification scheme of proof methods as outlined below (see §7.3.2). Rarely used variants are incorporated into the main methods via separate arguments and options. Thus Isar offers the user some choice from a collection of automated tools that is relatively easy to oversee.

Note that Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] and DECLARE [Syme, 1997a] [Syme, 1998] [Syme, 1999] essentially provide only a single builtin proof tool for solving terminal obligations, where only some additional parameters (facts and hints) may be given by the user. This policy takes the issue of "declarative theorem proving" very seriously, since it liberates the user from specifying operational details.

On the other hand, practical experience with Isabelle/Isar has shown that the overall robustness of mechanized proof processing may be improved by leaving the user some choice of adequate means for specific problems. As a general rule of thumb, simpler methods should be preferred over complex ones, e.g. single rules or plain rewriting instead of heavy combinations of simplification and classical reasoning. This is essentially Occam's razor applied to proof methods.

Consequently, the resulting Isar proof texts are not only processed more quickly, but also provide additional clues about the level of complexity of particular problems. For example, DECLARE would essentially refer to "**by** *auto*" all the time, where Isar admits to differentiate the complexity of proofs ranging over "**by** *blast*" and "**by** *simp*", down to "**..**" and "**.**".

A completely different issue than incorporating existing proof methods is that of actually writing new ones. High-level specification of general proof procedures is

definitely outside the present scope of Isar. Note that the present Isar framework natively supports derived rules of the meta-logic (§5.2.5), which already covers many practically relevant situations where one might have considered specific methods a requirement at first sight.

The current Isabelle/Isar implementation requires some ML programming to include new proof methods [Wenzel, 2001a], although this is usually only done by those who start an object-logic from scratch, or extend an existing one significantly. An end-user environment like Isabelle/HOL is already sufficiently equipped to support a broad range of Isabelle/Isar applications, without demanding the user to extend the system by separate ML programming. The generic tools of Isabelle turn out as sufficiently flexible in practice. For example, normalization wrt. associativity and commutativity may be easily achieved via ordered rewriting with Isabelle's Simplifier [Paulson, 2001b] without requiring any ad-hoc programming as proposed in [Zammit, 1999b].

On the other hand, high-level representations of proof procedures could complement the primary Isar proof language in a useful manner. Approaches that internalize procedures into the base logic (e.g. [Barendregt *et al.*, 1995] [Ruys, 1999]) could be used within the Isar setting on top of a suitable object-logic with support for reflection principles. An alternative may be to extend the primary Isar language itself to cover method definitions as well, similar to the specification format for proof procedures proposed in [Arkoudas, 1998].

7.3.2 Basic types of proof methods

The Isar framework already includes a few fundamental proof methods (§3.3.2) that either refer to the key concept of higher-order backchaining in various ways (*this*, *rule*, *assumption*), or perform normalization wrt. a collection of meta-level equalities (*unfold*, *fold*). Some further methods have already been provided as a more abstract view on top of these, notably *cases* and *induct* (§5.4).

In contrast, automated proof methods may perform numerous basic inferences inside, usually guided by additional rule declarations given in the present context (with classification according to the intended use rules as simplifications, introductions, eliminations etc.). Furthermore, automated procedures are generally prone to require long runtime (due to advanced heuristics and extensive proof search), or may even diverge on particular problem classes.

In order to arrive at a reasonably policy to incorporate the multitude of existing Isabelle tactics into Isar, we now introduce the following classification scheme of automated proof methods.

First of all, methods may either *simplify* or *solve* goals. Here "simplification" means that individual goals may be replaced by any number of new ones (e.g. by normalization wrt. a certain collection of rules, splitting structural case expressions, or logical decompositions according to standard introductions and eliminations). In reality, the resulting statements might appear to the user as

more complicated rather than simpler. Unlike internal simplification tactics in Isabelle [Paulson, 2001b], Isar requires simplification to make actual progress, i.e. unchanged goal configurations cause the process to fail altogether. The former behaviour may be recovered via the "*?*" method combinator (cf. §3.3.2). "Solving" a goal makes it disappear altogether; note that this may influence other goals due to instantiation of schematic variables in the proof state.

Furthermore, we distinguish the range of goals addressed by a method, covering either the *head* (first subgoal) or *all* subgoals. Note that arbitrary goal addressing is considered inappropriate for Isar proof texts. Reconsidering the standard terminal proof pattern of "**by** m_1 m_2" (§3.3.3), Isar methods either occur initially (like m_1) with only a single subgoal present, or terminally (like m_2) where all remaining goals essentially need to be covered simultaneously. Proof scripts emulated within Isabelle/Isar may refer to the improper commands of **prefer** and **defer** to shuffle subgoals at will, see also [Wenzel, 2001a].

Methods

Isabelle/HOL provides the following collection of advanced proof methods for use in Isar, based on the Simplifier and Classical Reasoner inside.

method	focus	kind
simp	*simplifies head*	*simp*
simp-all	*simplifies all*	*simp*
clarify	*simplifies head*	*classical*
safe	*simplifies all*	*classical*
auto	*simplifies all*	*simp + classical*
blast	*solves head*	*classical*
force	*solves head*	*simp + classical*

In practice, the non-solving classical methods (*clarify* and *safe*) are mainly used for exploration, or tactic script emulation within Isar. Several further methods are required for porting of legacy scripts, like *fast*, *best*, *slow* etc. [Wenzel, 2001a]. New Isar applications may be conducted with a significantly reduced collection (essentially *simp*, *blast*, *auto*), since structured Isar proofs generally demand much less fiddling of the exact operational behaviour of automated methods.

Note that separate methods for "*solves all*" are not needed in practice, since repeated application of "*solves head*" versions already have a similar effect. E.g. we may refer to "*blast*+" instead of providing a separate version of *blast-all*.

Methods of the "*simplifies all*" focus (most notably *auto*) are apt to notorious inconveniences when used in *unstructured* proof scripts. The problem is that *auto* touches all existing subgoals, replacing them by a probably large number of new ones that do not necessarily look that "simple" after all. The goal configurations tends to be blown up and lose any previous structure, making it hard to continue the script by further method applications afterwards. Proper Isar proofs do *not* suffer from this problem, since modularity of structured

proof processing limits the scope of methods to isolated sub-proofs (which are statically delimited in the text). Thus the common Isar idiom of "**by** *auto*" essentially puts *auto* in a "sandbox". So this discipline turns a potentially hazardous procedure into a viable multi-purpose proof tool.

Attributes

The above collection of proof methods are subject to specific context information for standard rule declarations, using the *data* field within the global *theory* or the local proof *context* (cf. §3.2.2 and §3.2.3). Rule declarations are managed via separate attributes as outlined below, see also [Wenzel, 2001a].

attribute	description
kind *simp*:	
simp add	add simplification rule
simp del	delete ditto
split add	add case-splitting rule
split del	delete ditto
cong add	add congruence rule
cong del	delete ditto
kind *classical*:	
intro	add introduction rule
elim	add elimination rule
dest	add destruction rule
rule del	delete classical rule
kind *simp* + *classical*:	
iff add	add simultaneous simplification + introduction/elimination
iff del	delete ditto

Attributes "*xxx add*" may be abbreviated as *xxx*. The classical declarations of *intro*/*elim*/*dest* may include "!" or "*?*" modifiers to indicate especially high or low priorities, respectively; similarly for "*iff?*" versus *iff*. See also [Paulson, 2001b] and [Wenzel, 2001a] for further details on the exact role of these different kinds of rule declarations. Note that such subtleties are mainly relevant for Isar writers only. Readers may just choose to ignore certain annotations, but merely observe that rules may get used somehow later on.

Arguments and facts

Method arguments admit to augment rule declarations for immediate use, fully analogous to the above attributes. For example, consider a proof context declaration like "**note** *a* [*intro*] **and** *b* [*elim*]" versus a method invocation like

"**by** (*blast intro*: *a elim*: *b*)". See [Wenzel, 2001a] for concrete syntax specifications.

Simplifier arguments provide a separate short-hand: the method specification "(*simp add*: *a*)" refers to the recurrent pattern of adding simplification rules, instead of the more logical (but cumbersome) forms of "(*simp simp*: *a*)" or even "(*simp simp add*: *a*)". The special case "(*simp only*: *a*)" refers to simplification without any other rules than *a*. This achieves another "strength reduction" of Isabelle's Simplifier that turns out as quite handsome in many applications, since unexpected simplifications from the global context are excluded.

Facts offered to automated methods via forward chaining (involving Isar's **then** primitive) are not treated specifically, but are merely inserted into the goal configuration as local premises just before the actual proof procedure is invoked. It is very important to note that any further facts available in the Isar proof context are *never* used silently. This policy enables the writer to indicate the "relevance of facts" explicitly in the proof text, which turns out as an important aspect of readability of formal documents that involve notoriously obscure automated reasoning tools (see also the detailed discussion in §7.5.2).

7.4 The main Isabelle/HOL library

Theoretical expositions of formal logic usually include an extensive discussion of the primitive axiomatic basis, together with meta-theoretical properties of the deductive systems, and maybe some model theory. In contrast, applied logic should eventually reach a stage where such foundational details become much less important than the concrete theory environment offered to end-users. The very core principles may still have some impact on how advanced concepts may get implemented, especially definitional packages (§7.2) and proof methods (§7.3), but most users would not care too much about foundations as long as they get a system that fits their needs for realistic applications.

Concerning Isabelle/HOL [Nipkow *et al.*, 2001] the main library (of Isabelle99-2) consists of a DAG structure of 47 theory nodes, as presented in figure 7.2. Only 3 of these contribute to the axiomatic basis of the very HOL logic: *HOL* for the basic axioms according to [Gordon and Melham, 1993], *Set* for an isomorphic copy of predicates as type *'a set*, and *NatDef* for a type of individuals with the axiom of infinity. The remaining 44 theories are required to bootstrap a reasonable working environment, with definitions of pairs, disjoint sums, natural numbers, support for inductive sets, general datatypes, and recursive functions.

In fact, the dependency graph of definitional packages (cf. §7.2) is intertwined with that of theories given in figure 7.2: advanced packages usually require some basic concepts to start with. This easily leads into quasi-circular dependencies, which need to be untangled either by clever arrangement of basic concepts [Harrison, 1996a], or by special provisions of "inverted" definitions [Berghofer and Wenzel, 1999].

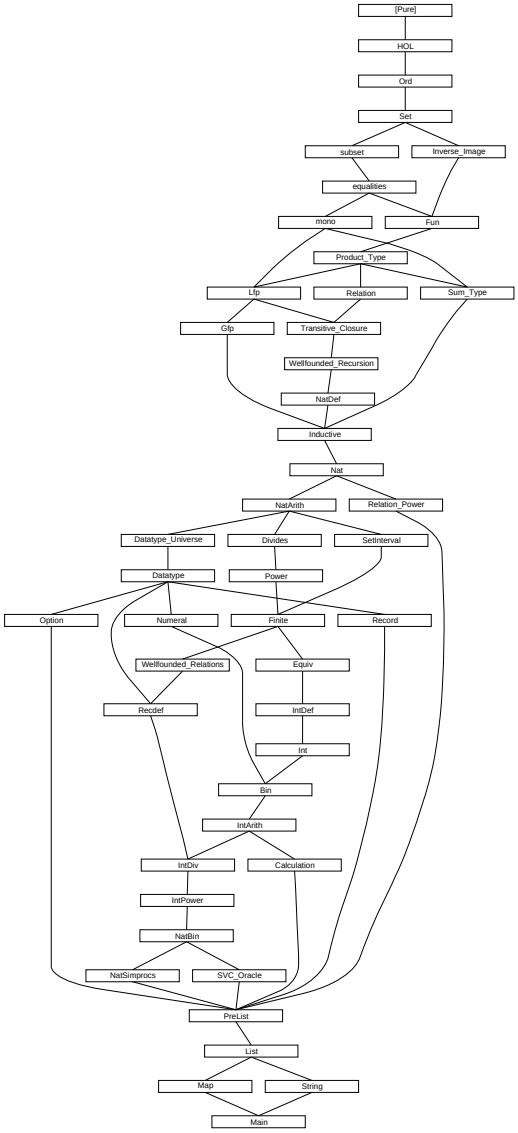

Figure 7.2: Main theory library of Isabelle/HOL

For example, **datatype** definitions require natural numbers internally, but the same type *nat* needs to be treated like a genuine datatype later on. This is achieved via **rep-datatype** in Isabelle/HOL, which takes an existing type with canonical theorems (covering freeness and induction) and retro-fits the remaining infrastructure of actual **datatype** definitions. Thus the primitively defined type *nat* may participate in **primrec** or **recdef** as well. It really behaves as if it had been defined like this.

datatype *nat* = 0 | *Suc nat*

The deeper reason for the substantial efforts of bootstrapping the main working environment of Isabelle/HOL lies in the key HOL methodology to start from very basic concepts only, and build up everything else by means of definitions and derivations within the formal system itself. Coq [Barras *et al.*, 1999] represents a slightly different approach, where the underlying type theory has already been equipped with powerful means of induction and recursion in the first place [Coquand and Paulin-Mohring, 1990] [Pfenning and Paulin-Mohring, 1990] [Paulin-Mohring, 1993]. These concepts have a clear justification within the meta-theory of the "Calculus of Inductive Constructions", which has been developed separately. In HOL such strong concepts would be constructed by reduction to primitive concepts of simply-typed set-theory, essentially at run-time of the theory processor. PVS [Owre *et al.*, 1996] follows yet another approach, where substantial parts of the library (and packages) are hardwired in the implementation, without any intrinsic formal justifications in the first place.

Independently of any such foundational issues of Isabelle/HOL, end-users are offered a comfortable working environment by means of the ultimate theory of *Main*. This provides of more faithful view on the "real" Isabelle/HOL logic, than the primitive basis of *HOL* given in the very beginning. The context provided by *Main*, with advanced induction and recursion being readily available, might even appeal to proponents of constructive type theory, since the details of classical reasoning (see §8.4), Hilbert's choice operator (see §8.5), and HOL type definitions (see §8.6) are largely hidden from sight.

Further user extensions of Isabelle/HOL usually do not involve complex bootstrapping issues anymore. Authors may build their theories according to the natural dependencies arising from definitions and proofs. In the same vein the supplemental library of generally useful Isabelle/HOL theories [Bauer *et al.*, 2001] has been developed outside of the monolithic part of main Isabelle/HOL.

7.5 Discussion

7.5.1 Theory specifications versus proofs

Speaking in terms of primitive logical concepts (e.g. chapter 2), specifications and proofs are the two fundamental means to achieve new results in a certain

context. Experience with existing theorem proving environments shows that applied logic demands a proper methodology for interaction of these different aspects. Several definitional concepts require separate coverage of proof obligations, such as **typedef** (§7.1.2) and **instance** (§7.2.4) in Isabelle/HOL.

Interestingly, the type theory tradition has been able to unify both concepts of specification and proof within the same framework of typed λ-calculus (e.g. see the exposition in [Barendregt and Geuvers, 2001]). Nevertheless, existing type theory provers may still offer a differentiated view to the user: Coq [Barras *et al.*, 1999] clearly separates its specification language "Gallina" from the tactical proof language, although both are in principle able to produce the same kind of λ-terms. In practice, Coq occasionally requires decisions by the user if a "proof" is better presented as a "specification" in certain situations. A typical example are side-conditions stemming from partial algebraic operations like division, see also the related discussion in §9.4.2, and [Geuvers *et al.*, 2000].

In the original LCF/HOL tradition [Gordon, 2000] specifications and proofs have been treated differently in the basic logic [Pitts, 1993], but essentially uniformly from the perspective of users. The view of such systems is based on an ML top-level loop, with a collection of abstract datatype constructors for primitive inference rules and definitional extensions alike. This ML interface of HOL generally tends towards bare-bones construction of theory and theorem objects. For example, the non-emptiness obligation required by the type definition primitive (see also §8.6) needs to be stated as a separate theorem beforehand, feeding the result into the subsequent definition stage as an ML value by hand.

Another commonly encountered HOL technique is to make advanced definitional packages solve proof obligations fully automatically inside. Here the implementation (in ML) typically exploits the specific form of proof problems emerging from a certain class of specifications, e.g. inductive datatypes of a particular form. In Isabelle/HOL, the same approach appears in **inductive** (§7.2.1), **datatype** (§7.2.1), **primrec** (and **recdef**) (§7.2.2), and **record** (§7.2.3).

Such an automated setup works reasonably well for schematic proof obligations arising from well-defined specification schemes (e.g. **datatype** and **primrec**). Things become slightly more difficult for more general mechanisms like TFL [Slind, 1996] [Slind, 1997] (**recdef** in Isabelle/HOL). Here the proof obligations (of termination etc.) may be arbitrarily complex, depending on the actual recursive function specification given by the user.

As already pointed out earlier (§7.2.2), **recdef** refers to a number of standard automated proof tools that may only be controlled indirectly via hints, i.e. previously established facts with an indication of their role in the automated process (cf. the Isar attributes *recdef-simp*, *recdef-cong*, *recdef-wf* given in [Wenzel, 2001a]). That arrangement essentially follows the original HOL tradition of feeding previous theorems into definitional mechanisms, but mediates the use of auxiliary facts through complex proof procedures. In practice, this amounts to mostly obscure automated reasoning, supplanting even the existing techniques

of interactive tactical proving in HOL.

Recent work on well-founded recursion in Coq [Balaa and Bertot, 2000] exhibits analogous problems of incorporating separate proof obligations in non-trivial definitional patterns. According to the unified approach of type theory, proofs of side-conditions may *in principle* included directly as λ-terms in the specification text. This naive approach turns out as slightly impractical due to the complexity of the proof objects encountered in well-founded recursion. [Balaa and Bertot, 2000] propose a specific Coq mechanism to achieve a more abstract presentation of a restricted class of well-founded recursive definitions.

PVS [Owre *et al.*, 1996] follows a different approach to side-conditions arising from specifications. It does *not* consider proofs as first-class members of theory developments in the first place, but treats everything uniformly as genuine proof obligations to be managed separately (including type-checking conditions due to predicate subtyping of logical statements, termination of recursive definitions, or even plain theorem statements). Here "proofs" are managed dynamically in special files, the actual theory merely consists of definitions and statements. A theory is marked as finished once that all obligations have been covered interactively by the user. There is also specific support for change management with automatic replay old proof scripts behind the scenes; individual failures of previous proofs are marked accordingly.

PVS is positioned as a "prototype verification system", where users may develop formal descriptions that are explored by means of interactive proof checking (and model-checking), in order to exhibit bugs in their designs. Consequently, "proofs" are treated as mere necessities that pop up dynamically and need to accommodated by specific tool to achieve maximum comfort for users. The concept of proofs as independent static representations of formal reasoning work conducted by users is not considered important here.

Specifications and readable proofs

From the Isar perspective, the issue of incorporating proofs into specification mechanisms need to be reconsidered from the primary view of formal document construction, rather than primitive logical issues or even particular system organization. In informal mathematics, definitions and proofs may be freely intermixed in the text without further ado. For example, operations on quotient structures typically demand "well-definedness" proofs, establishing congruence properties modulo a certain equivalence relation.

Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] successfully follows this basic observation of mathematical practice. Some Mizar specification schemes include proof texts to cover separate obligations. The subsequent example has been taken from article #676 (`DYNKIN`) of [Mizar library]. It illustrates an indirect definition by giving a (unique) characterization of the intended entity (see the `it` expression in the `means` part below).

```
definition
```

```
 let Omega be non empty set;
 let f be SetSequence of Omega;
 let X be Subset of Omega;
 func seqIntersection(X,f) -> SetSequence of Omega
 means :def_seqInt:
 for n holds it.n=X∩ f.n;
 existence
 proof
  ...
 end;
 uniqueness
 proof
  ...
 end;
end;
```

Here the `existence` and `uniqueness` keywords indicate specific goal statements required by that particular definitional scheme. The proof bodies are formulated in terms of the existing Mizar proof language.

In Isar we intend to follow essentially the same idea of incorporating the existing proof language into theory specification mechanisms in a modular fashion. Some care is required to accommodate the open-ended nature of Isar syntax (§3.2.1 and §3.2.4), and the incremental way of Isar proof processing (§3.2.3). Our general technique is to split the definitional mechanism into three stages as indicated below. The existing **theorem** command of Isar (§3.2.1) turns out as sufficiently flexible to provide the link between these stages, just by virtue of the attribute specification that may be included in the initial result specification.

1. The *preamble* may perform initial preparations (and syntactic checks of the specification text), before commencing an ordinary Isar proof by posing a new claim at the theory level.

2. The *proof* proceeds incrementally by existing Isar language elements given by the user. The package does not have any immediate control about the shape of that proof, but may have included some special support via initial context declarations, such as term abbreviations or auxiliary facts.

   For example, we may ensure that canonical **proof** steps do the "right thing" to commence a manual proof in practice, by virtue of appropriate rule declarations. Trivial situations should work out in a single automated step as well, e.g. "**by** *simp*".

3. The *postamble* acquires the result established by the user and eventually performs the actual theory extension.

Note that we prefer to include only a single Isar proof into specification mechanisms (without loss of generality). Multiple obligations may be easily represented either within the object-logic (e.g. using plain $\wedge$), or may be achieved

simultaneously according to the same principles underlying **obtain** (cf. §5.3). This policy avoids pseudo-goal statements, like `existence` and `uniqueness` exhibited in the Mizar example above. Conical Isar proof patterns may involve plain **show** elements (in the order preferred by the writer).

We illustrate the general Isar technique of mixed specifications and proofs by **typedef** of Isabelle/HOL (cf. §7.1.2). This is actually a derived theory command, based on the **theorem** primitive as indicated below. Here A refers to the representing set specified by the user; the package demands a non-emptiness statement $\vdash \exists x.\ x \in A$ to complete the actual "definition" (see also §8.6).

theorem [*perform-typedef*]: $\exists x.\ x \in A$ (**is** $\exists x.\ x \in ?set$)

The user may now fill in an Isar proof immediately in the text, for example by a canonical **proof** step, which in turn demands a concrete existential witness to be given in the subsequent sub-proof (by virtue of $\exists$ introduction).

In order to accommodate this common pattern, the **typedef** package has already provided a suitable term abbreviation in the initial goal statement (the actual name is that of the type definition, cf. §7.1.2).

proof
  show $a \in ?set$ ⟨*proof*⟩
qed

Having finished that proof, the result $\vdash \exists x.\ x \in A$ is fed into the special attribute *perform-typedef*, which essentially performs a type definition according to the primitive in Gordon/HOL implementations already discussed before [Gordon and Melham, 1993] [Gordon, 2000]. Recall that a theory attribute may be an arbitrary function $\mathit{theory} \times \mathit{theorem} \to \mathit{theory} \times \mathit{theorem}$ (cf. the interpretation $\mathcal{A}$ in §3.2.3). Here we simply let *perform-typedef* augment the theory by the type definition primitive (passing the non-emptiness fact).

The **instance** specification of Isabelle/Isar (§7.2.4) works along similar lines. Here the initial claim would be the logical reflection of type arities or class inclusion statements [Wenzel, 1997]. Since standard class introduction rules are available as global introduction patterns, the user may again use a standard **proof** step to reduce this raw goal to a number of subgoals corresponding exactly to the class axioms that need to be established for the particular instantiation at hand.

It remains to be seen whether the same technique of combining advanced specification mechanisms with readable proofs scales up to more sophisticated packages like TFL [Slind, 1996] [Slind, 1997]. A successful reformulation of **recdef** within Isabelle/Isar should provide both a better understanding of the different stages of its internal proof process, as well as significantly improved user convenience. Isar proof contexts provide far more infrastructure to lay out complex proof obligations than primitive goal states used in TFL so far.

7.5.2 Proof methods and relevance of facts

Consider the following recurrent Isar proof pattern of establishing an intermediate fact by an atomic proof step.

from $\vec{a}$ **have** C **by** m

A text fragment like this is able to communicate several important aspects of formal reasoning to the reader, including usage of existing facts $\vec{a}$, the result statement C, and the proof method m. The above Isar phrase appears to be quite intelligible due its pseudo-natural language reading. On the other hand, we may raise the key question if the formal meaning of the corresponding sequence of Isar/VM transitions encountered here (cf. §3.2.3) is actually consistent with such an informal reading of the text. In principle, the real behavior of formal proof texts could belie readers, for example if names of primitive elements and concrete syntax would have been chosen badly to begin with.

We call a portion of Isar proof text *faithful* iff its general perception by readers coincides with the actual formal meaning. Apparently, faithfulness of texts is not easily determined exactly, it heavily depends on the general expectations of particular readers, and may be up to a broad range of interpretations of the "real" meaning.

Concerning the overall language design of Isar (cf. chapter 3), great care has been taken to achieve a high degree of faithfulness of proof texts by default, following our common style of writing. Nevertheless, with the open design of Isar and its highly compositional nature that allows different kinds of language elements to be combined in numerous ways, it is hard to rule out semantic anomalies altogether. The Isar proof writer may certainly produce non-sense by full intention, but we would not like to take away this freedom at the cost of the inflexibility of a more restricted language.

On the other hand, anomalies should not creep into Isar proofs by accident. Reconsidering the three aspects of the elementary pattern above, we first observe that Isar ensures proper treatment of the first two at the least: referenced facts $\vec{a}$ are guaranteed to have been established beforehand, and the explicit statement C is always that of the real result produced eventually. Only the proof method m involved here might be apt to unexpected effects, since arbitrary inferences may be performed inside.

Even basic *rule* steps (§3.3.2) occasionally have unexpected effects; higher-order backchaining (§2.4) is not a completely trivial operation after all. The subsequent example demonstrates how proof texts based on single inferences may already belie the reader.

assume ab: $A \wedge B$
assume C

$\vdots$

from *ab* **have** *C* **..**
— non-faithful proof text!

Apparently, this proof is slightly inappropriate, since it fools the reader to think that the fact $ab = \vdash A \wedge B$ contributes to the result $\vdash C$. In reality, the local assumptions A and B emerging from the initial elimination have been *ignored*. Even worse, a different assumption was used implicitly, according to the builtin notion of finished Isar goal configurations up to proof-by-assumption (§3.2.3). The sane proof is documented in further detail below.

from *ab* **have** *C*
proof
assume *A* **and** *B*
— ignored
have *C* **.**
— finished by assumption
qed

Fortunately, the danger of producing such faulty texts by accident is not very high in practice, provided the writer takes minimal care in proof composition. Note that individual applications of rules (and assumptions) are quite easy to trace in interactive development, especially in conjunction with the Proof General interface [Proof General] [Aspinall, 2000].

The situation changes substantially with arbitrary automated proof methods (cf. §7.3). Here we encounter the fundamental problem of *relevance of facts* in automated reasoning. Common proof procedures (e.g. the first-order tableau prover implemented in Isabelle's *blast* [Paulson, 1999]) operate on a goal configuration that may include any number of premises considered as "axioms" for the purpose of the present proof search. The general behavior of common search procedures critically depends on this collection of facts included in the problem. Both by adding or removing local facts, the search space (and runtime) may grow or shrink considerably, just as overall failure or divergence of the procedure. Afterwards it is usually hard to tell which particular facts have really made the difference.

Ideally, proof procedures would include comprehensive diagnostic information about the relevance of facts encountered on the internal path of automated proof search. Unfortunately, this poses a fundamental problem for existing techniques of automated reasoning, both in theory and practice. Note that most automated tools would not even record the internal success path in terms of basic inference of the underlying logic, which makes it hard acquire full representations of (primitive) proof objects. Recording relevance of any internal inference steps would be even more difficult.

Isar does not meddle with specific issues of implementing automated proof tools properly, but offers the following discipline in order to keep the critical dependency on local facts under control of the proof writer. As a general rule, "advanced" methods may never refer to previous facts from the proof contexts

themselves (such as assumptions or local results), but only include those that have been explicitly highlighted by the proof writer (either by forward chaining via **then**, or as separate arguments in the method specification).

This simple policy turns out as quite robust in practice. While it is not possible to make sure that all facts actually do participate in building the result, readers will know that *at most* these local facts may have been relevant in that particular automated step.

Now reconsider our basic proof patterns involving a heavily automated method.

from $\vec{a}$ **have** C **by** *blast*

Independent of the exact course of reasoning performed inside of *blast*, the text is able to tell the reader that only the local facts $\vec{a}$ may immediately contribute to the result of C.

Note that global declarations of additional rules (e.g. *intro*, *elim*, *dest* rules for *blast*, cf. §7.3) are a slightly different issue. For syntactic reasons, global rules may not refer to local entities of a particular proof context. Nonetheless, a badly designed theory library may cause rather unexpected behaviour of automated tools in its own right.

Another beneficial effect of restricting the use of local facts is that Isar goal configurations are kept relatively small. This essentially enforces a limited form of compositionality of automated tools wrt. augmenting proof contexts (cf. the related discussion in §6.4.3): additional assumptions and facts are not included in advanced method invocations in the first place, so they cannot disturb their behavior in uncouth manners.

In contrast, unstructured tactical proof scripts tend to suffer severely from goal states that are crowded by excessive facts. The basic paradigm of tactical proving is to apply consecutive transformations of goals, until a solved form is reached eventually (cf. §3.2.3 and §4.2.3). Here all information is accumulated in one big goal state, which routinely makes automated methods choke in realistic applications. For this reason, tactical systems usually provide special tools to tune goal states in an ad-hoc fashion, e.g. `thin_tac` in traditional Isabelle [Paulson, 2001b].

On the other hand, structured proof systems like Isabelle/Isar have become more scalable without any special provisions required. See also the experience reported in chapter 10, especially concerning the run-time behavior of non-trivial Isar proof texts (see §10.7.2). The particular technique of "big-step" reasoning via degenerate calculational proof schemes (cf. §6.4.3) essentially provides another perspective to the relevance problem of facts, which may have been collected over several sections of Isar proof text.

Chapter 8

Example: Higher-Order Logic

We reconsider foundational issues of the HOL logic, ranging from the pure framework of minimal higher-order logic to slightly more exotic features of classical HOL used in practice, including Hilbert's choice operator and type definitions.

From the Isar perspective, this formal development serves as a realistic example of studying issues of pure logic, conducted at the level of higher-order abstract syntax and derived rules. We do not yet use Isabelle/HOL here, but discuss its very foundations within the pure background theory of Isabelle/Isar.

8.1 Minimal Higher-Order Logic

theory *Higher-Order-Logic* = *Pure*:

8.1.1 Simply-typed lambda-terms

The language of simply-typed λ-terms is represented within the background theory by the well-known approach of *higher-order abstract syntax* [Pfenning and Elliott, 1988].

We first introduce a subclass *type* for the types of our language of λ-terms. The types o of propositions and $\to$ of functions operate on this class.

classes *type* $\subseteq$ *logic*
defaultsort *type*

typedecl o
arities o :: *type*

typedecl $('a, 'b) \to$ (**infixr** 0)
arities $\to$ :: (*type*, *type*) *type*

The signatures of abstraction and application of the object-language are declared as follows. We also state β- and η-conversion rules as equality axioms.

consts
$Abs :: ('a \Rightarrow 'b) \Rightarrow 'a \to 'b$ (**binder** $\boldsymbol{\lambda}$ 5)
$App :: ('a \to 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixl** $\cdot$ 500)
axioms
beta-conv: $(\boldsymbol{\lambda}\, x.\ f(x)) \cdot a \equiv f(a)$
eta-conv: $(\boldsymbol{\lambda}\, x.\ f \cdot x) \equiv f$

The above representation basically provides a separate copy of the syntactic background λ-calculus, which happens to have the same conversion laws, but here we have been able to keep the underlying equational theory under our own control. Consequently, the use of conversions of the object-language will be exposed explicitly in our proofs.

Explicit reasoning with conversions over the basic syntax happens to be part of the present meta-theoretical study on HOL. In contrast, an actual application environment (chapter 7) would identify its syntax with the existing framework as much as possible, in order to spare users unnecessary formal noise. Then β- and η-conversion would take place implicitly in builtin operations, especially those of higher-order unification and back-chaining (cf. §2.4).

8.1.2 Basic logical connectives

With the abstract syntax of simply-typed λ-terms available, actual logical properties of the language are now represented via a *truth judgment* that coerces the type o of object-level statements into propositions *prop* of the framework. This is the canonical way to represent object-logics within Isabelle's meta-logic [Paulson, 1989] [Paulson, 1990]. Note that the concrete syntax leaves the *Trueprop* coercion implicit as a special notation of category *prop*.

judgment
$Trueprop :: o \Rightarrow prop$ (- 5)

Implication and universal quantification (over arbitrary types) are now declared with syntax and characteristic rules as follows.

consts
$imp :: o \to o \to o$
$All :: ('a \to o) \to o$

syntax
$\text{-}imp :: o \Rightarrow o \Rightarrow o$ (**infixr** $\longrightarrow$ 25)
$\text{-}All :: idt \Rightarrow o \Rightarrow o$ ((3$\forall$ -./ -) [0, 10] 10)
translations
$A \longrightarrow B \ \rightleftharpoons\ imp \cdot A \cdot B$
$\forall x.\ A \ \rightleftharpoons\ All \cdot (\boldsymbol{\lambda}\, x.\ A)$

axioms
impI [*intro*]: $(A \Longrightarrow B) \Longrightarrow A \longrightarrow B$
impE [*dest*, *trans*]: $A \longrightarrow B \Longrightarrow A \Longrightarrow B$
allI [*intro*]: $(\bigwedge x.\ P(x)) \Longrightarrow \forall x.\ P(x)$
allE [*dest*]: $\forall x.\ P(x) \Longrightarrow P(a)$

The representation of quantified propositions $\forall x.\ A$ as $All \cdot (\boldsymbol{\lambda} x.\ A)$ follows the earliest tradition of higher-order logic [Church, 1940]. The same principle may be applied to similar "binders" as well, such as $\sum i < n.\ e(i)$. This casual handling of variable binding via λ-abstraction is one of the key virtues of higher-order abstract syntax [Pfenning and Elliott, 1988].

The axioms given above may be understood as *introduction* rules for the truth judgment. This view is sufficient to study the resulting logical system at the level of primitive and derived rules. On the other hand, there is *no induction* rule available to reason about derivability of *Trueprop* statements in an exhaustive manner. In particular, admissible rules are presently beyond our reach.

An established way to model a logical system with both introduction rules and induction would be to define derivability as an inductive set (§7.2.1) over the basic syntax. This would enable the full range of meta-theoretical studies (including completeness issues etc.), leaving behind the handsome approach of higher-order abstract syntax. There has been ongoing work on combining higher-order abstract syntax directly with induction principles [Despeyroux *et al.*, 1997] [Hofmann, 1999], although the resulting systems do have their own complexities, both in theory and practice.

8.2 Extensional equality

The axiomatic base of minimal logic is now extended by a particular kind of equality. The following declarations introduce "=" as an extensional equivalence relation that coincides with logical equivalence. We are careful to introduce atomic axioms only (cf. the treatment of the pure framework in §2.2); it is easy to derive proper rules as well, which are more useful in practice.

consts
equal :: $'a \to 'a \to o$

syntax
-*equal* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** = 50)
translations
$x = y \rightleftharpoons$ *equal* $\cdot\ x \cdot y$

axioms
refl [*intro*]: $x = x$

subst-ax: $x = y \longrightarrow P \cdot x \longrightarrow P \cdot y$
ext-ax: $(\forall x.\ f \cdot x = g \cdot x) \longrightarrow f = g$
iff-ax: $(A \longrightarrow B) \longrightarrow (B \longrightarrow A) \longrightarrow A = B$

theorem *subst*: $x = y \Longrightarrow P(x) \Longrightarrow P(y)$
proof −
note *subst-ax*
also assume $x = y$
also assume $P(x)$ **hence** $(\boldsymbol{\lambda} x.\ P(x)) \cdot x$ **by** (*unfold beta-conv*)
finally have $(\boldsymbol{\lambda} x.\ P(x)) \cdot y$ **. thus** $P(y)$ **by** (*unfold beta-conv*)
qed

theorem *ext* [*intro*]: $(\bigwedge x.\ f \cdot x = g \cdot x) \Longrightarrow f = g$
proof −
note *ext-ax*
also assume $\bigwedge x.\ f \cdot x = g \cdot x$ **hence** $\forall x.\ f \cdot x = g \cdot x$ **..**
finally show $f = g$ **.**
qed

theorem *iff* [*intro*]: $(A \Longrightarrow B) \Longrightarrow (B \Longrightarrow A) \Longrightarrow A = B$
proof −
note *iff-ax*
also assume $A \Longrightarrow B$ **hence** $A \longrightarrow B$ **..**
also assume $B \Longrightarrow A$ **hence** $B \longrightarrow A$ **..**
finally show $A = B$ **.**
qed

We are ready to derive the basic properties of "=" as a congruence of the underlying term language. Symmetry and transitivity are consequences of reflexivity and substitution. We also declare standard transitivity rules for calculational reasoning (cf. chapter 6).

theorem *sym* [*elim*]: $x = y \Longrightarrow y = x$
proof −
assume $x = y$
thus $y = x$ **by** (*rule subst*) (*rule refl*)
qed

lemma [*trans*]: $x = y \Longrightarrow P(y) \Longrightarrow P(x)$
by (*rule subst*) (*rule sym*)

lemma [*trans*]: $P(x) \Longrightarrow x = y \Longrightarrow P(y)$
by (*rule subst*)

theorem *trans* [*trans*]: $x = y \Longrightarrow y = z \Longrightarrow x = z$
by (*rule subst*)

Next we derive the congruences for application and abstraction. The former is a simple consequence of reflexivity and substitution. The latter involves additional properties of the term language, namely extensionality and β-conversion.

theorem *app-cong* [*intro*]: $f = g \Longrightarrow x = y \Longrightarrow f \cdot x = g \cdot y$
proof −
  assume $f = g$ **hence** $f \cdot x = g \cdot x$ **by** (*rule subst*) (*rule refl*)
  also assume $x = y$ **hence** $g \cdot x = g \cdot y$ **by** (*rule subst*) (*rule refl*)
  finally show *?thesis* **.**
qed

lemma [*intro*]: $x = y \Longrightarrow f \cdot x = f \cdot y$
  by (*rule app-cong*) (*rule refl*)

theorem *abs-cong* [*intro*]: $(\bigwedge x.\ f(x) = g(x)) \Longrightarrow (\boldsymbol{\lambda}\, x.\ f(x)) = (\boldsymbol{\lambda}\, x.\ g(x))$
proof −
  assume *eq*: $\bigwedge x.\ f(x) = g(x)$
  show *?thesis*
  proof
    fix x **from** *eq* **have** $f(x) = g(x)$ **.**
    thus $(\boldsymbol{\lambda}\, x.\ f(x)) \cdot x = (\boldsymbol{\lambda}\, x.\ g(x)) \cdot x$ **by** (*unfold beta-conv*)
  qed
qed

It is very important to note that in our formulation of equality within the framework substitution had been chosen as primitive, with emerging congruences as derived rules. Otherwise, we would have required induction to establish substitution as an admissible rule, which is not possible here.

Finally we complete the characterization of "=" as logical equivalence. The following two eliminations are basic consequences of substitution in HOL.

theorem *iff*$_1$ [*elim*]: $A = B \Longrightarrow A \Longrightarrow B$
  by (*rule subst*)

theorem *iff*$_2$ [*elim*]: $A = B \Longrightarrow B \Longrightarrow A$
  by (*rule subst*) (*rule sym*)

8.3 Further connectives

8.3.1 Definitions

Having unrestricted quantification over propositions available in minimal higher-order logic, we may now introduce the standard set of logical connectives in a purely definitional manner. The representation of standard connectives based only on $\longrightarrow$ and $\forall$ closely follows established traditions of higher-order logic and type theory. As a general rule of thumb, the definition of a derived connective such as $\exists$ follows its canonical elimination rule.

consts
  false :: *o* (⊥)
  true :: *o* (⊤)
  not :: *o* → *o*
  conj :: *o* → *o* → *o*
  disj :: *o* → *o* → *o*
  Ex :: ('*a* → *o*) → *o*

syntax
  -*not* :: *o* ⇒ *o* (¬ - [40] 40)
  -*not-equal* :: '*a* ⇒ '*a* ⇒ '*a* (**infixl** ≠ 50)
  -*conj* :: *o* ⇒ *o* ⇒ *o* (**infixr** ∧ 35)
  -*disj* :: *o* ⇒ *o* ⇒ *o* (**infixr** ∨ 30)
  -*Ex* :: *idt* ⇒ *o* ⇒ *o* ((3∃ -./ -) [0, 10] 10)
translations
  ¬ *A* ⇌ *not* · *A*
  x ≠ *y* ⇌ ¬ (*x* = *y*)
  A ∧ *B* ⇌ *conj* · *A* · *B*
  A ∨ *B* ⇌ *disj* · *A* · *B*
  ∃ *x*. *P* ⇌ *Ex* · (**λ** *x*. *P*)

defs
  false-def: ⊥ ≡ ∀ *A*. *A*
  true-def: ⊤ ≡ ⊥ ⟶ ⊥
  not-def: *not* ≡ **λ** *A*. *A* ⟶ ⊥
  conj-def: *conj* ≡ **λ** *A* *B*. ∀ *C*. (*A* ⟶ *B* ⟶ *C*) ⟶ *C*
  disj-def: *disj* ≡ **λ** *A* *B*. ∀ *C*. (*A* ⟶ *C*) ⟶ (*B* ⟶ *C*) ⟶ *C*
  Ex-def: *Ex* ≡ **λ** *P*. ∀ *C*. (∀ *x*. *P* · *x* ⟶ *C*) ⟶ *C*

8.3.2 Derived rules

Based on the above definitions, the usual characteristic rules for standard logical connectives may now be derived as follows.

theorem *falseE* [*elim*]: ⊥ ⟹ *A*
proof (*unfold false-def*)
  assume ∀ *A*. *A*
  thus *A* **..**
qed

theorem *trueI* [*intro*]: ⊤
proof (*unfold true-def*)
  show ⊥ ⟶ ⊥ **..**
qed

theorem *notI* [*intro*]: (*A* ⟹ ⊥) ⟹ ¬ *A*
proof (*unfold not-def*)
  assume *A* ⟹ ⊥
  hence *A* ⟶ ⊥ **..**
  thus (**λ** *A*. *A* ⟶ ⊥) · *A* **by** (*unfold beta-conv*)

qed

theorem *notE* [*elim*]: $\neg\ A \Longrightarrow A \Longrightarrow B$
proof (*unfold not-def*)
  assume $(\boldsymbol{\lambda}\, A.\ A \longrightarrow \bot) \cdot A$
  hence $A \longrightarrow \bot$ **by** (*unfold beta-conv*)
  also assume A
  finally have $\bot$ **..**
  thus B **..**
qed

lemma *notE′*: $A \Longrightarrow \neg\ A \Longrightarrow B$
  by (*rule notE*)

lemmas *contradiction* = *notE notE′* — proof by contradiction in any order

theorem *conjI* [*intro*]: $A \Longrightarrow B \Longrightarrow A \wedge B$
proof (*unfold conj-def*)
  assume *a*: A **and** *b*: B
  have $\forall\, C.\ (A \longrightarrow B \longrightarrow C) \longrightarrow C$
  proof
    fix C **show** $(A \longrightarrow B \longrightarrow C) \longrightarrow C$
    proof
      assume $A \longrightarrow B \longrightarrow C$
      also note *a*
      also note *b*
      finally show C **.**
    qed
  qed
  thus $(\boldsymbol{\lambda}\, A\ B.\ \forall\, C.\ (A \longrightarrow B \longrightarrow C) \longrightarrow C) \cdot A \cdot B$
    by (*unfold beta-conv*)
qed

theorem *conjE* [*elim*]: $A \wedge B \Longrightarrow (A \Longrightarrow B \Longrightarrow C) \Longrightarrow C$
proof (*unfold conj-def*)
  assume $(\boldsymbol{\lambda}\, A\ B.\ \forall\, C.\ (A \longrightarrow B \longrightarrow C) \longrightarrow C) \cdot A \cdot B$
  hence *c*: $\forall\, C.\ (A \longrightarrow B \longrightarrow C) \longrightarrow C$ **by** (*unfold beta-conv*)
  assume $A \Longrightarrow B \Longrightarrow C$
  thus C
  proof *this*
    show A
    proof −
      from *c* **have** $(A \longrightarrow B \longrightarrow A) \longrightarrow A$ **..**
      also have $A \longrightarrow B \longrightarrow A$
      proof
        assume A
        thus $B \longrightarrow A$ **..**
      qed
      finally show *?thesis* **.**
    qed

```
    show B
    proof −
      from c have (A ⟶ B ⟶ B) ⟶ B ..
      also have A ⟶ B ⟶ B
      proof
        show B ⟶ B ..
      qed
      finally show ?thesis .
    qed
  qed
qed

theorem disjI1 [intro]: A ⟹ A ∨ B
proof (unfold disj-def)
  assume a: A
  have ∀ C. (A ⟶ C) ⟶ (B ⟶ C) ⟶ C
  proof
    fix C show (A ⟶ C) ⟶ (B ⟶ C) ⟶ C
    proof
      assume A ⟶ C
      also note a
      finally have C .
      thus (B ⟶ C) ⟶ C ..
    qed
  qed
  thus (λ A B. ∀ C. (A ⟶ C) ⟶ (B ⟶ C) ⟶ C) · A · B
    by (unfold beta-conv)
qed

theorem disjI2 [intro]: B ⟹ A ∨ B
proof (unfold disj-def)
  assume b: B
  have ∀ C. (A ⟶ C) ⟶ (B ⟶ C) ⟶ C
  proof
    fix C show (A ⟶ C) ⟶ (B ⟶ C) ⟶ C
    proof
      show (B ⟶ C) ⟶ C
      proof
        assume B ⟶ C
        also note b
        finally show C .
      qed
    qed
  qed
  thus (λ A B. ∀ C. (A ⟶ C) ⟶ (B ⟶ C) ⟶ C) · A · B
    by (unfold beta-conv)
qed

theorem disjE [elim]: A ∨ B ⟹ (A ⟹ C) ⟹ (B ⟹ C) ⟹ C
proof (unfold disj-def)
```

```
  assume (λ A B. ∀ C. (A ⟶ C) ⟶ (B ⟶ C) ⟶ C) · A · B
  hence c: ∀ C. (A ⟶ C) ⟶ (B ⟶ C) ⟶ C by (unfold beta-conv)
  assume r₁: A ⟹ C and r₂: B ⟹ C
  show C
  proof −
    from c have (A ⟶ C) ⟶ (B ⟶ C) ⟶ C ..
    also have A ⟶ C
    proof
      assume A thus C by (rule r₁)
    qed
    also have B ⟶ C
    proof
      assume B thus C by (rule r₂)
    qed
    finally show ?thesis .
  qed
qed

theorem exI [intro]: P(a) ⟹ ∃ x. P(x)
proof (unfold Ex-def)
  assume a: P(a)
  have ∀ C. (∀ x. (λ x. P(x)) · x ⟶ C) ⟶ C
  proof
    fix C show (∀ x. (λ x. P(x)) · x ⟶ C) ⟶ C
    proof
      assume ∀ x. (λ x. P(x)) · x ⟶ C
      hence (λ x. P(x)) · a ⟶ C ..
      hence P(a) ⟶ C by (unfold beta-conv)
      also note a
      finally show C .
    qed
  qed
  thus (λ P. ∀ C. (∀ x. P · x ⟶ C) ⟶ C) · (λ x. P(x))
    by (unfold beta-conv)
qed

theorem exE [elim]: ∃ x. P(x) ⟹ (⋀x. P(x) ⟹ C) ⟹ C
proof (unfold Ex-def)
  assume (λ P. ∀ C. (∀ x. P · x ⟶ C) ⟶ C) · (λ x. P(x))
  hence c: ∀ C. (∀ x. (λ x. P(x)) · x ⟶ C) ⟶ C by (unfold beta-conv)
  assume r: ⋀x. P(x) ⟹ C
  show C
  proof −
    from c have (∀ x. (λ x. P(x)) · x ⟶ C) ⟶ C ..
    also have ∀ x. (λ x. P(x)) · x ⟶ C
    proof
      fix x show (λ x. P(x)) · x ⟶ C
      proof
        assume (λ x. P(x)) · x
        hence P(x) by (unfold beta-conv)
```

```
      thus C by (rule r)
    qed
  qed
  finally show ?thesis .
 qed
qed
```

8.4 Classical logic

There are many ways to characterize classical logic. The following axiom (and the corresponding derived rule) express classical reasoning in a very explicit way: in order to show a proposition one may just assume its negation!

```
axioms
  classical-ax: (¬ A ⟶ A) ⟶ A

theorem classical: (¬ A ⟹ A) ⟹ A
proof −
  note classical-ax
  also assume ¬ A ⟹ A hence ¬ A ⟶ A ..
  finally show A .
qed
```

Peirce's Law is a similar well-known characterization of classical logic, which uses only implication in its statement. Observing that $\neg\ A$ acts like $A \longrightarrow \bot$ (we have even defined it that way), Peirce's Law can be understood as a formal generalization of the *classical-ax* statement, with $\neg\ A$ represented by $A \longrightarrow B$ for an arbitrary proposition B.

```
theorem Peirce's-Law: ((A ⟶ B) ⟶ A) ⟶ A
proof
  assume a: (A ⟶ B) ⟶ A
  show A
  proof (rule classical)
    assume ¬ A
    have A ⟶ B
    proof
      assume A
      thus B by (rule contradiction)
    qed
    with a show A ..
  qed
qed
```

Some alternative classical rules may be derived as follows: *double-negation*, *tertium-non-datur*, and *classical-cases* (which is particularly useful in practice).

theorem *double-negation*: $\neg\ \neg\ A \Longrightarrow A$
proof −
  assume $\neg\ \neg\ A$
  show A
  proof (*rule classical*)
    assume $\neg\ A$
    thus *?thesis* **by** (*rule contradiction*)
  qed
qed

theorem *tertium-non-datur*: $A \vee \neg\ A$
proof (*rule double-negation*)
  show $\neg\ \neg\ (A \vee \neg\ A)$
  proof
    assume $\neg\ (A \vee \neg\ A)$
    have $\neg\ A$
    proof
      assume A **hence** $A \vee \neg\ A$ **..**
      thus $\bot$ **by** (*rule contradiction*)
    qed
    hence $A \vee \neg\ A$ **..**
    thus $\bot$ **by** (*rule contradiction*)
  qed
qed

theorem *classical-cases*: $(A \Longrightarrow C) \Longrightarrow (\neg\ A \Longrightarrow C) \Longrightarrow C$
proof −
  assume r_1: $A \Longrightarrow C$ **and** r_2: $\neg\ A \Longrightarrow C$
  from *tertium-non-datur* **show** C
  proof
    assume A
    thus *?thesis* **by** (*rule* r_1)
  next
    assume $\neg\ A$
    thus *?thesis* **by** (*rule* r_2)
  qed
qed

Apparently, the above rules entail each other in the given order. One may even close the circle back to the *classical* rule (without using that rule in the proof, of course). Thus we illustrate the well-known fact that any of these rules may serve as a complete characterization of classical reasoning itself; cf. the comprehensive exposition in [Thompson, 1991].

lemma $(\neg\ A \Longrightarrow A) \Longrightarrow A$
proof −
  assume r: $\neg\ A \Longrightarrow A$

```
  show A
  proof (rule classical-cases)
    assume A thus A .
  next
    assume ¬ A thus A by (rule r)
  qed
qed
```

8.5 Hilbert's choice operator

Hilbert's choice operator ε (which is called *Some* below) takes an arbitrary predicate and delivers an unspecified element belonging to its extension; the result is completely unknown for the empty predicate. Using the usual binder syntax, we write $SOME\ x.\ P(x)$ for selection from the predicate $\boldsymbol{\lambda} x.\ P(x)$.

```
consts
  Some :: ('a → o) → 'a

syntax
  -Some :: idt ⇒ o ⇒ o    ((3SOME -./ -) [0, 10] 10)
translations
  SOME x. P  ⇌  Some · (λ x. P)

axioms
  some-ax: P · a ⟶ P · (SOME x. P · x)

theorem someI: P(a) ⟹ P(SOME x. P(x))
proof −
  note some-ax
  also assume P(a) hence (λ x. P(x)) · a by (unfold beta-conv)
  finally have (λ x. P(x)) · (SOME x. (λ x. P(x)) · x) .
  thus ?thesis by (unfold beta-conv)
qed
```

Hilbert's choice operator is occasionally presented as a slightly mysterious mechanism, which appears to allow elements to be picked even from the empty set! On the other hand, it may be seen as just another total higher-order function that happens to be somewhat underspecified within the formal framework of HOL. This view fits indeed very well into the general "totality" approach of HOL, where any well-formed expression is treated as properly defined without demanding a unique interpretation.

Nevertheless, most of mainstream mathematics is usually content with unique descriptions, by selecting elements from singleton sets. In fact, this is the most common use of Hilbert's choice in HOL as well. The formulation given in [Andrews, 1986] even includes an operator for unique descriptions only. The following derived rule covers that special case in a practically useful manner, it shows

how to "evaluate" a choice expression by exhibiting a unique witness.

theorem *some-equality*:
  $P(a) \Longrightarrow (\bigwedge x.\ P(x) \Longrightarrow x = a) \Longrightarrow (SOME\ x.\ P(x)) = a$
proof −
  assume r: $\bigwedge x.\ P(x) \Longrightarrow x = a$
  assume $P(a)$ **hence** $P(SOME\ x.\ P(x))$ **by** (*rule someI*)
  thus $(SOME\ x.\ P(x)) = a$ **by** (*rule r*)
qed

Occasionally, the general form of Hilbert's choice may be found useful nonetheless. The example of rational numbers in chapter 9 includes a simple theory of quotient types, where the general choice principle is used to pick "default" elements from equivalence classes (which are only unique in terms of the underlying equivalence relation), see §9.2.3.

8.6 Concrete types and type definitions

So far, our presentation of the basic HOL logic has only provided the basic type o of propositions (which degenerates into the boolean values for classical systems), and functions $'a \to 'b$ over arbitrarily complex types $'a$ and $'b$.

In order to achieve a sufficiently rich environment to represent mathematical concepts, it is customary to axiomatize another type i as an infinite collection of "individuals". Subsequently we use the common characterization of infinity as "there is an injection that is not a surjection".

typedecl i
arities $i :: type$

axioms
  i-infinite: $\exists f :: i \to i.\ (\forall x.\ \forall y.\ f \cdot x = f \cdot y \longrightarrow x = y) \wedge (\exists z.\ \forall x.\ f \cdot x \neq z)$

In principle, the types of i, o, and $'a \to 'b$ are already sufficient to represent common mathematical notions within HOL, according to its inherent expressive power; e.g. see the exposition of [Andrews, 1986] for the original formulation of [Church, 1940]. For example, f and z acquired by the *i-infinite* property above turn out as suitable representatives of successor and zero, respectively; thus the natural numbers may be considered as a subset of type i.

On the other hand, the potential of HOL types is somewhat diminished by this elementary approach; types would merely be used to achieve a syntactic "ranking" of objects, in order to guarantee consistency of the system (this happens to be Church's original intention, after an untyped system has failed due to Russel's paradox). Thus HOL would essentially just be treated like a restricted version of untyped set-theory.

Church's original formulation of the "Simple Theory of Types" [Church, 1940]

was later rediscovered by Gordon as a useful basis for computer-science applications, with interactive development of machine-checked formal proof [Gordon, 1985a] [Gordon, 1985b] [Gordon and Melham, 1993] [Gordon, 2000].

At that point HOL has acquired an improved treatment of types, supporting arbitrary type constructors and schematic polymorphism. Furthermore, the mechanism of HOL type definitions was devised, in order to be able to introduce new types in a disciplined manner; the HOL tradition generally rejects arbitrary axiomatizations by end-users.

8.6.1 Basic characterization of type definitions

HOL type definitions state axioms to identify a non-empty subset of an existing type with a new type. This is expressed in a set-theoretic manner via two bijections *rep* and *abs* as specified below. The subsequent property of *type-definition* will be stated as an axiom for any new type $'a$, which has been represented as a non-empty subset of an existing type $'b$. Here we identify sets over $'b$ with predicates $'b \to o$; Isabelle/HOL uses a separate type of sets (chapter 7).

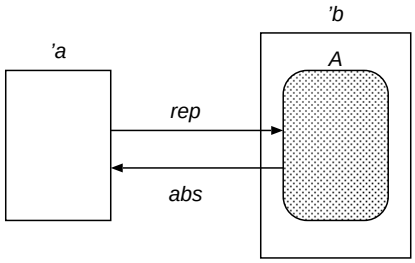

Figure 8.1: HOL type definition

constdefs
  $type\text{-}definition :: ('a \to 'b) \to ('b \to 'a) \to ('b \to o) \to o$
  $type\text{-}definition \equiv \boldsymbol{\lambda}\, rep\ abs\ A.$
    $(\forall x.\ A \cdot (rep \cdot x)) \wedge$
    $(\forall x.\ abs \cdot (rep \cdot x) = x) \wedge$
    $(\forall y.\ A \cdot y \longrightarrow rep \cdot (abs \cdot y) = y)$

This compact axiomatization may be decomposed into separate theorems of *rep*, *rep-inverse*, and *abs-inverse* as follows.

lemma *type-definitionE* [*elim*]:
  $type\text{-}definition \cdot rep \cdot abs \cdot A \Longrightarrow$
    $(A \cdot (rep \cdot x) \Longrightarrow$
      $abs \cdot (rep \cdot x) = x \Longrightarrow$

```
    A · y ⟶ rep · (abs · y) = y ⟹ C) ⟹ C
  (is - ⟹ (?rep(x) ⟹ ?rep-inverse(x) ⟹ ?abs-inverse(y) ⟹ -) ⟹ -)
proof (unfold type-definition-def beta-conv)
  assume a: (∀ x. ?rep(x)) ∧ (∀ x. ?rep-inverse(x)) ∧ (∀ y. ?abs-inverse(y))
  assume r: ?rep(x) ⟹ ?rep-inverse(x) ⟹ ?abs-inverse(y) ⟹ C
  from a have b: (∀ x. ?rep-inverse(x)) ∧ (∀ y. ?abs-inverse(y)) ..
  from a have ∀ x. ?rep(x) .. hence ?rep(x) ..
  moreover from b have ∀ x. ?rep-inverse(x) .. hence ?rep-inverse(x) ..
  moreover from b have ∀ y. ?abs-inverse(y) .. hence ?abs-inverse(y) ..
  ultimately show C by (rule r)
qed

theorem rep: type-definition · rep · abs · A ⟹ A · (rep · x)
  by (rule type-definitionE)

theorem rep-inverse: type-definition · rep · abs · A ⟹ abs · (rep · x) = x
  by (rule type-definitionE)

theorem abs-inverse: type-definition · rep · abs · A ⟹
  A · y ⟹ rep · (abs · y) = y
proof −
  assume type-definition · rep · abs · A
  hence A · y ⟶ rep · (abs · y) = y ..
  also assume A · y
  finally show ?thesis .
qed
```

It is important to note that HOL types need to be treated as inherently non-empty, due to the most basic inference rules. In particular, $(\forall x.\ P(x)) \longrightarrow (\exists x.\ P(x))$ is a theorem for any type of x. The proof given below works, because fixing an arbitrary element of some syntactic HOL type does not pose any additional constraint on a result that does not mention that element afterwards.

```
theorem (∀ x::'a::type. P(x)) ⟶ (∃ x. P(x))
proof
  fix any
  assume ∀ x. P(x)
  hence P(any) ..
  thus ∃ x. P(x) ..
qed
```

The fundamental non-emptiness of HOL types may be also observed in the behavior of basic theory extensions (cf. §2.3). In particular, constants of *arbitrary* type may be declared at any time, without affecting fundamental meta-theoretical properties of the theory in a relevant manner (the way that HOL is used in practice does not admit any strong properties here in the first place).

Hilbert's choice operator (cf. §8.5) provides yet another way to achieve witness terms for arbitrary type schemes, although this slightly more exotic construction is not really required to observe these effects, which are already present in the most pure formulation of higher-order logic (cf. §2.2).

We see that HOL type definitions necessarily require non-emptiness of the representing set to be established beforehand. This condition is already sufficient to preserve certain semantical properties of the resulting axiom scheme to be considered as "definitional" [Pitts, 1993]; the class of standard models of classical HOL ensures type universes to be closed by forming non-empty subsets.

On the other hand, HOL type definitions do *not* share further meta-theoretical properties of plain constant definitions (cf. §2.3). As demonstrated in [Wenzel, 1997], type definitions are not syntactically conservative; thus consistency need not be preserved either, since the notion of "standard models" underlying the argument in [Pitts, 1993] is incomplete wrt. the deductive system of HOL.

It is very important to note that this incomplete view on HOL is merely an effect of the particular set-theoretic interpretation required by [Pitts, 1993] in order to make the **typedef** primitive appear as a definitional concept. The very higher-order nature of HOL does *not* make it apt to incompleteness. In fact, [Henkin, 1950] shows completeness of the original formulation of [Church, 1940]. The proof may use essentially the same techniques as for propositional or first-order logic, cf. the detailed exposition in [Andrews, 1986].

8.6.2 Derived rules of type definitions

The primitive axioms of HOL type definitions are quite cumbersome to use in practice. The following theorems express a higher-level view: injections amount to simplification rules for equality, and surjections yield (degenerate) rules for cases and induction (both for the new type and the original set).

theorem *rep-inject*: *type-definition* · *rep* · *abs* · *A* ⟹
  (*rep* · *x* = *rep* · *y*) = (*x* = *y*)
proof −
  assume *a*: *type-definition* · *rep* · *abs* · *A*
  show *?thesis*
  proof
    assume *rep* · *x* = *rep* · *y*
    hence *abs* · (*rep* · *x*) = *abs* · (*rep* · *y*) **..**
    also from *a* **have** *abs* · (*rep* · *x*) = *x* **by** (*rule rep-inverse*)
    also from *a* **have** *abs* · (*rep* · *y*) = *y* **by** (*rule rep-inverse*)
    finally show *x* = *y* **.**
  next
    assume *x* = *y*
    thus *rep* · *x* = *rep* · *y* **..**
  qed
qed

theorem *abs-inject*: *type-definition* · *rep* · *abs* · *A* ⟹
  A · *x* ⟹ *A* · *y* ⟹ (*abs* · *x* = *abs* · *y*) = (*x* = *y*)
proof −
  assume *a*: *type-definition* · *rep* · *abs* · *A*
  assume *x*: *A* · *x* **and** *y*: *A* · *y*
  show *?thesis*
  proof
    assume *abs* · *x* = *abs* · *y*
    hence *rep* · (*abs* · *x*) = *rep* · (*abs* · *y*) **..**
    also from *a* **and** *x* **have** *rep* · (*abs* · *x*) = *x* **by** (*rule abs-inverse*)
    also from *a* **and** *y* **have** *rep* · (*abs* · *y*) = *y* **by** (*rule abs-inverse*)
    finally show *x* = *y* **.**
  next
    assume *x* = *y*
    thus *abs* · *x* = *abs* · *y* **..**
  qed
qed

theorem *rep-cases*: *type-definition* · *rep* · *abs* · *A* ⟹
  A · *y* ⟹ (⋀*x*. *y* = *rep* · *x* ⟹ *C*) ⟹ *C*
proof −
  assume *a*: *type-definition* · *rep* · *abs* · *A* **and** *y*: *A* · *y*
  assume (⋀*x*. *y* = *rep* · *x* ⟹ *C*)
  thus *C*
  proof *this*
    from *a* **and** *y* **have** *rep* · (*abs* · *y*) = *y* **by** (*rule abs-inverse*)
    thus *y* = *rep* · (*abs* · *y*) **..**
  qed
qed

theorem *abs-cases*: *type-definition* · *rep* · *abs* · *A* ⟹
  (⋀*y*. *x* = *abs* · *y* ⟹ *A* · *y* ⟹ *C*) ⟹ *C*
proof −
  assume *a*: *type-definition* · *rep* · *abs* · *A*
  assume ⋀*y*. *x* = *abs* · *y* ⟹ *A* · *y* ⟹ *C*
  thus *C*
  proof *this*
    from *a* **have** *abs* · (*rep* · *x*) = *x* **by** (*rule rep-inverse*)
    thus *x* = *abs* · (*rep* · *x*) **..**
    from *a* **show** *A* · (*rep* · *x*) **by** (*rule rep*)
  qed
qed

theorem *rep-induct*: *type-definition* · *rep* · *abs* · *A* ⟹
  A · *y* ⟹ (⋀*x*. *P*(*rep* · *x*)) ⟹ *P*(*y*)

proof −
  assume *a*: *type-definition* · *rep* · *abs* · *A*
  assume $\bigwedge x.\ P(rep \cdot x)$ **hence** $P(rep \cdot (abs \cdot y))$ **.**
  also assume $A \cdot y$ **with** *a* **have** $rep \cdot (abs \cdot y) = y$ **by** (*rule abs-inverse*)
  finally show $P(y)$ **.**
qed

theorem *abs-induct*: *type-definition* · *rep* · *abs* · $A \Longrightarrow$
  $(\bigwedge y.\ A \cdot y \Longrightarrow P(abs \cdot y)) \Longrightarrow P(x)$
proof −
  assume *r*: $\bigwedge y.\ A \cdot y \Longrightarrow P(abs \cdot y)$
  assume *a*: *type-definition* · *rep* · *abs* · *A*
  hence $A \cdot (rep \cdot x)$ **by** (*rule rep*)
  hence $P(abs \cdot (rep \cdot x))$ **by** (*rule r*)
  also from *a* **have** $abs \cdot (rep \cdot x) = x$ **by** (*rule rep-inverse*)
  finally show $P(x)$ **.**
qed

In the real Isabelle/HOL environment (chapter 7), the above cases and induct rules may get used implicitly by the *cases* and *induct* proof methods (cf. the general proof patterns given in §7.1.2). Technically, we treat HOL type definitions like a hybrid of inductive set and datatype (cf. §7.2.1), although there is no recursion involved here.

The resulting high-level setup of Isabelle/Isar reduces the formal noise involved in detailed type abstraction and representation issues to a reasonable level. Thus HOL type definitions turn out as an actually useful mechanisms for end-user applications. So far the raw **typedef** was generally considered too cumbersome.

See chapter 9 for a concrete application that uses Isabelle/HOL's **typedef** primitive effectively in Isabelle/Isar.

end

8.7 Discussion: Isar techniques

Technically speaking, the present formulation of higher-order logic within the basic framework of Isabelle/Pure has been quite similar to the initial example of first-order logic given in chapter 4. Consequently, we have employed similar techniques of reasoning with plain natural deduction in single steps, without any automated proof tools available yet. As our derivations in HOL have been slightly more "realistic" than the ones of FOL before, we have referred to a few additional techniques in order to keep the tedium of manual reasoning at a reasonable level.

Note that this lack of advanced proof tools exhibits an inherent issue of "bootstrapping" new object-logics formulated within the pure framework. Automated

proof tools (e.g. Isabelle's tableau prover *blast* [Paulson, 1999]) typically require a number of auxiliary theorems for their internal setup; obviously, these need to be derived by simpler means beforehand.

As far as the Isabelle tradition of building up object-logics is concerned, such early bootstrapping stages tend to consist of a slightly unstructured collection of proof scripts, with heavy use of ad-hoc "automation" simulated by tactic combinators (especially for repeating and alternative choices of scripts).

The present Isabelle/Isar application demonstrates that decent proofs may be performed from the very start of a new object-logic. We argue that even such "primitive" theories deserve proper treatment of formal proofs, in order to enable interested readers to understand the details of building formal-reasoning environments from scratch. We think that there is no need to hold up the general perception of the primitive concepts of mechanized reasoning systems as slightly arcane matter, where it would be equally possible to employ lucid declarative techniques.

Subsequently, we point out a few notable techniques to accommodate the lack of automated proof tools, as encountered in the present HOL formulation.

Calculating with implication

Implication "$\longrightarrow$" is frequently encountered in the formal text, stemming from atomic axioms and definitions of object-level connectives. Since we wish to derive proper rules formulated by the meta-level connective "$\Longrightarrow$", we need to treat implication accordingly, typically by means of the *modus ponens* rule $\vdash A \longrightarrow B \Longrightarrow A \Longrightarrow B$. As it happens, this rule directly fits into the paradigm of calculational reasoning (cf. chapter 6). Given an implication $A \longrightarrow B$ as current calculational result, we may continue the chain by adding also A, in order to conclude B in the next step; with iterated (curried) implications, the latter would be an implication, too, so we may continue the chain as expected.

For example, this technique is exhibited in theorem *iff*. Recall that the statement of *iff-ax* has been $(A \longrightarrow B) \longrightarrow (B \longrightarrow A) \longrightarrow A = B$.

theorem *iff* [*intro*]: $(A \Longrightarrow B) \Longrightarrow (B \Longrightarrow A) \Longrightarrow A = B$
proof −
  note *iff-ax*
  also assume $A \Longrightarrow B$ **hence** $A \longrightarrow B$ **..**
  also assume $B \Longrightarrow A$ **hence** $B \longrightarrow A$ **..**
  finally show $A = B$ **.**
qed

The overall effect of calculating with implication is similar to (restricted) state-oriented scripting techniques, as encountered in the `Intros` command of Coq [Barras *et al.*, 1999], for example. The `assume` element of Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] is essentially the same as

well (cf. §4.2.4 and §5.5.1). So certain forms of Isar calculations may be used as a disciplined replacement for operational goal transformations; the implicit compositions performed with calculational sequences internally are restricted to the set of transitivity rules declared beforehand (cf. chapter 6).

Unlike "$\longrightarrow$" elimination as encountered in the present example, the analogous "$\forall$" elimination rule is *not* quite suitable for calculational reasoning, because it has only one premise. In simply-typed HOL, the syntactic typing of an element is not treated like a logical judgment (cf. §2.2). Nevertheless, one could consider to calculate with set-bounded quantifiers as available in Isabelle/HOL (chapter 7), by using the rule $\vdash \forall x \in A.\ P(x) \Longrightarrow a \in A \Longrightarrow P(a)$ (cf. §6.3.1).

Compact presentation of multiple conclusions

The complete lack of automated reasoning tools can be felt more severely in §8.6.1 where we have derived 3 projections of the *type-definition* statement. The individual proofs would basically proceed in the same manner, by eliminating quantifiers and projecting conjunctions. This rather repetitious kind of inferences is singled out into the separate lemma *type-definitionE*. Here multiple simultaneous results are expressed in the canonical fashion, just like conjunction would be represented within minimal logic: recall that $\bigwedge C.\ (X \Longrightarrow Y \Longrightarrow Z \Longrightarrow C) \Longrightarrow C$ acts just like the conjunction of X, Y, Z.

lemma *type-definitionE* [*elim*]:
  type-definition · *rep* · *abs* · *A* ⟹
    (*A* · (*rep* · *x*) ⟹
      abs · (*rep* · *x*) = *x* ⟹
      A · *y* ⟶ *rep* · (*abs* · *y*) = *y* ⟹ *C*) ⟹ *C*
  (**is** - ⟹ (*?rep*(*x*) ⟹ *?rep-inverse*(*x*) ⟹ *?abs-inverse*(*y*) ⟹ -) ⟹ -)

The proof body is kept reasonably abstract by using term abbreviations. We refrain from "cheating" via ad-hoc proof scripts, but take the issue of structured proof texts seriously, despite this rather boring instance.

proof (*unfold type-definition-def beta-conv*)
  assume *a*: (∀ *x*. *?rep*(*x*)) ∧ (∀ *x*. *?rep-inverse*(*x*)) ∧ (∀ *y*. *?abs-inverse*(*y*))
  assume *r*: *?rep*(*x*) ⟹ *?rep-inverse*(*x*) ⟹ *?abs-inverse*(*y*) ⟹ *C*
  from *a* **have** *b*: (∀ *x*. *?rep-inverse*(*x*)) ∧ (∀ *y*. *?abs-inverse*(*y*)) **..**
  from *a* **have** ∀ *x*. *?rep*(*x*) **.. hence** *?rep*(*x*) **..**
  moreover from *b* **have** ∀ *x*. *?rep-inverse*(*x*) **.. hence** *?rep-inverse*(*x*) **..**
  moreover from *b* **have** ∀ *y*. *?abs-inverse*(*y*) **.. hence** *?abs-inverse*(*y*) **..**
  ultimately show *C* **by** (*rule r*)
qed

Extracting rules from basic definitions (both introductions and eliminations) is a recurrent pattern in applications of the natural deduction framework of Isabelle [Paulson and Nipkow, 1994]. As such transition are mostly formal bookkeeping

tasks only, the proofs are best represented as an atomic step, using suitable proof tools.

In real Isabelle/HOL (chapter 7) one would typically refer to the idiom of "**by** (*unfold type-definition-def*) *blast*", although the full power of classical tableau proving is actually overkill here. Nevertheless, that tool happens to be available in Isabelle [Paulson, 1999], and is very quick on a broad range of logical proof problems. In a non-classical setting, one would probably take care to offer a much simpler tool to achieve this kind of basic formal rearrangement of connectives.

Chapter 9

Example: Rational numbers

We present a theory of rational numbers based on the canonical representation of equivalence classes over pairs of integers. The standard algebraic laws of fields are proven as well.

This development covers both the domain of abstract algebraic structures, as well as mathematical modeling involving concrete representations. The basic ideas of the present formalization are close to traditional textbook expositions, although we employ a few advanced techniques specific to Isabelle/HOL such as axiomatic type classes and type abstractions. We observe that Isabelle/Isar is able to handle mathematical applications adequately, much larger ones have already been performed elsewhere.

9.1 Motivation

Classical mathematics appears to be the canonical domain for non-trivial applications of structured proof languages, mostly due to the Mizar system [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] that has been successfully applied in collecting a large body of formalized mathematics [Mizar library]. Mathematicians know the value of "real proofs", which are *readable* representations of logical arguments essentially by definition. So it is not surprising that a theorem proving environment built by mathematicians strongly emphasizes structured proof texts. In contrast, systems built by computer-scientists tend to impose completely different views on existing problem spaces.

As far as Isabelle/Isar is concerned, we have already demonstrated that applications of pure logic may be presented adequately (cf. chapter 4 and chapter 8). The general techniques of chapter 5 and chapter 6 carry over to classical mathematics just as well.

The irrationality of $\sqrt{2}$ is a popular example to explain the concept of (informal)

mathematical proof in the first place. The subsequent Isar text presents a fully formal version at a reasonable level of abstraction (using the standard theories about integer division and real numbers from Isabelle/HOL, cf. chapter 7). Here we establish a generalized statement for any prime number (including 2).

theorem $x^2 = \mathit{real}\ p \Longrightarrow p \in \mathit{prime} \Longrightarrow x \notin \mathbb{Q}$
proof
  assume *x-sqrt*: $x^2 = \mathit{real}\ p$
  assume *p-prime*: $p \in \mathit{prime}$
  hence *p*: $1 < p$ **by** (*simp add*: *prime-def*)
  assume $x \in \mathbb{Q}$
  then obtain *m n* **where**
    n: $n \neq 0$ **and** *x-rat*: $|x| = \mathit{real}\ m\ /\ \mathit{real}\ n$ **and** *gcd*: $\mathit{gcd}\ (m, n) = 1$ **..**
  have *eq*: $m^2 = p * n^2$
  proof −
    from *n x-rat* **have** $\mathit{real}\ m = |x| * \mathit{real}\ n$ **by** *simp*
    hence $\mathit{real}\ (m^2) = x^2 * \mathit{real}\ (n^2)$ **by** (*simp split*: *abs-split*)
    also from *x-sqrt* **have** $\ldots = \mathit{real}\ (p * n^2)$ **by** *simp*
    finally show *?thesis* **..**
  qed
  have *p dvd m* ∧ *p dvd n*
  proof
    from *eq* **have** *p dvd* m^2 **..**
    with *p-prime* **show** *p dvd m* **by** (*rule prime-dvd-square*)
    then obtain *k* **where** $m = p * k$ **..**
    with *eq* **have** $p^2 * k^2 = p * n^2$ **by** (*simp add*: *mult-ac*)
    with *p* **have** $n^2 = p * k^2$ **by** *simp*
    hence *p dvd* n^2 **..**
    with *p-prime* **show** *p dvd n* **by** (*rule prime-dvd-square*)
  qed
  hence *p dvd gcd* (*m*, *n*) **..**
  with *gcd* **have** *p dvd* 1 **by** *simp*
  hence $p \leq 1$ **by** (*simp add*: *dvd-imp-le*)
  with *p* **show** *False* **by** *simp*
qed

Just as in the applications of basic logic covered so far (cf. chapter 4 and chapter 8), the Isar text focuses on explicit statements plus inherent structure indicating the composition of individual items; explicit references to local and global facts are kept at a minimum, detailed proof method specifications are avoided.

Compared to a similar proof given in article #593 of [Mizar library], the Isar version requires less explicit references to individual facts (both local ones and from the library), and less purely technical steps to make the proof work out properly in terms of the builtin automated checker.

Any "realistic" application of human-readable proof construction is confronted with the general issue of slightly unexpected behavior of the automated proof

tools involved. From the perspective of composing high-level proof texts, automated tools tend to be rather "uneven" in the sense that some very simple reasoning steps just fail, while other more complex ones happen to work out immediately. Apparently, this is a general problem of the kind of automated reasoning tools available today; a similar experience has been observed in long-term experience with Mizar [Rudnicki, 1992] [Trybulec, 1993].

End-users are usually not interested in the details of automated reasoning procedures, so such technical happenstance is slightly unsatisfactory. Mizar's builtin notion of "obvious inferences" [Rudnicki, 1987] is relatively simple, compared to the standards of existing automated reasoning technology; the Simplifier of Isabelle/HOL (which is the only advanced proof tool used in the example above) is conceptually even simpler. Nevertheless, the exact behavior is already hard to predict in practice, and the situation is usually much worse for heavily automated reasoning procedures, like *blast* and *force* in Isabelle/HOL (cf. §7.3).

So writing non-trivial proof texts involves some amount of experimentation, which Isabelle/Isar readily supports by interactive interpretation, probably including some "improper" proof commands (cf. chapter 3).

On the other hand, an important virtue of high-level proof checking is that ad-hoc use of automated reasoning may be replaced by well-defined concepts of structured proof composition in many situations. This principle can already be observed in Mizar (cf. §4.2.4), which separates processing of individual proof outline elements (`assume`, `thus`, etc.) from solving of problems in terminal position (via `by`), although both mechanisms are based on a particular view on classical first-order logic with a few non-intuitive effects.

In contrast, the basic machinery of Isar proof processing is essentially restricted to back-chaining of rules from minimal higher-order logic, involving higher-order unification as the only advanced concept (cf. chapter 2 and chapter 3). Any additional reasoning procedures are clearly marked by explicit method specifications in the text. The need for ad-hoc proof automation may be reduced even more by means of the derived Isar language elements of advanced natural deduction and calculational reasoning (cf. chapter 5 and chapter 6). Certainly, the latter concepts are completely redundant from the technical view of formal reasoning, but are indispensable to lay out realistic Isar proof texts adequately.

As a result, new users of Isabelle/Isar need to spend less time in learning to cope with automated tools, but may get started more quickly with a few common proof patterns based on distinctive elements. For example, meaningful applications of classical mathematics may already be performed by simple algebraic calculations, involving step-wise transformations of (in-)equalities via basic calculational **also**/**finally** (cf. chapter 6).

Interestingly, this very simple mode of reasoning in Isar turns out to be particularly difficult to emulate in established tactical theorem proving systems (cf. the experience reported in [Bauer and Wenzel, 2001]). This discrepancy might be related to the general bias of existing tactical proof tools (Coq, HOL,

Isabelle/HOL, etc.) towards heavy use of "logical reasoning" (including inductive techniques), rather than simple algebraic manipulations. Consequently, the traditional applications of these systems are more oriented towards computer-science than classical mathematics.

In contrast, the first non-trivial application of Isabelle/Isar has been from mathematics, namely the well-known Hahn-Banach theorem from functional analysis [Bauer, 1999] [Bauer, 2001a] [Bauer and Wenzel, 2000], based on the textbook exposition of [Heuser, 1986]. The complete formal development of [Bauer, 2001a] takes 63 printed pages; it includes basic facts about real vector spaces, subspaces, norms, ordering of functionals, and variations of the main theorem.

The original development of [Bauer, 1999] uses an early version of Isabelle/Isar, with several conveniences of the present environment still missing, such as advanced derived elements like generalized elimination (cf. §5.3). Nevertheless, the whole formal theory development has been completed as a C.S. Master's project within several weeks, including the time to get acquainted with the concepts of Isar and its implementation (which was not quite finished at that time).

The Hahn-Banach application certainly marks an important milestone in evaluating the concepts of Isar in practice. Although from the present day perspective, it turns out as a relatively simple exercise. Even more advanced mathematical applications have come within the range of Isabelle/Isar, now that its concepts and implementation have matured, and a large stock of standard proof techniques have been explored (cf. chapter 5 and chapter 6).

The present mathematical example is a relatively small and simple one. Subsequently we give a construction of rational numbers that demonstrates some techniques of both abstract algebraic structures and concrete mathematical modeling in Isabelle/Isar using the HOL logic. Our formal development uses specific concepts of Isabelle/HOL to some advantage, namely axiomatic type classes and type abstraction (cf. chapter 7). This nicely accommodates the notoriously cumbersome formal treatment of algebraic quotient structures; cf. [Harrison, 1996c] for a more traditional treatment within HOL-Light, involving some additional ML programming.

9.2 Quotient types

theory *Quotient* = *Main*:

We introduce the notion of generic quotient types over equivalence relations, together with definition principles for operations on quotients. Some basic ideas of this formalization stem from [Slotosch, 1997].

9.2.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations $\sim :: 'a \Rightarrow 'a \Rightarrow bool$.

axclass *eqv* $\subseteq$ *term*
consts
eqv :: ($'a$::*eqv*) $\Rightarrow$ $'a$ $\Rightarrow$ *bool* (**infixl** $\sim$ 50)

axclass *equiv* $\subseteq$ *eqv*
equiv-refl [*intro*]: $x \sim x$
equiv-trans [*trans*]: $x \sim y \Longrightarrow y \sim z \Longrightarrow x \sim z$
equiv-sym [*elim?*]: $x \sim y \Longrightarrow y \sim x$

The quotient type $'a$ *quot* consists of all equivalence classes over elements of the base type $'a$.

typedef $'a$ *quot* = $\{\{x.\ a \sim x\} \mid a::'a::eqv.\ True\}$
by *blast*

lemma *quotI* [*intro*]: $\{x.\ a \sim x\} \in quot$
by (*unfold quot-def*) *blast*

lemma *quotE* [*elim*]: $R \in quot \Longrightarrow (\bigwedge a.\ R = \{x.\ a \sim x\} \Longrightarrow C) \Longrightarrow C$
by (*unfold quot-def*) *blast*

Equivalence classes are the canonical representation of quotient elements.

constdefs
class :: $'a$::*equiv* $\Rightarrow$ $'a$ *quot* ($\lfloor$-$\rfloor$)
$\lfloor a \rfloor \equiv$ *Abs-quot* $\{x.\ a \sim x\}$

theorem *quot-exhaust*: $\exists a.\ A = \lfloor a \rfloor$
proof (*cases A*)
fix R **assume** R: A = *Abs-quot* R
assume $R \in quot$ **hence** $\exists a.\ R = \{x.\ a \sim x\}$ **by** *blast*
with R **have** $\exists a.\ A$ = *Abs-quot* $\{x.\ a \sim x\}$ **by** *blast*
thus *?thesis* **by** (*unfold class-def*)
qed

lemma *quot-cases* [*cases type*: *quot*]: $(\bigwedge a.\ A = \lfloor a \rfloor \Longrightarrow C) \Longrightarrow C$
by (*insert quot-exhaust*) *blast*

9.2.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

theorem *quot-equality* [*iff?*]: $(\lfloor a \rfloor = \lfloor b \rfloor) = (a \sim b)$
proof
assume $\lfloor a \rfloor = \lfloor b \rfloor$
hence $\{x.\ a \sim x\} = \{x.\ b \sim x\}$
by (*simp only*: *class-def Abs-quot-inject quotI*)
moreover have $a \sim a$ **..**
ultimately have $a \in \{x.\ b \sim x\}$ **by** *blast*

```
  hence b ∼ a by blast
  thus a ∼ b ..
next
  assume eqv: a ∼ b
  have {x. a ∼ x} = {x. b ∼ x}
  proof (rule Collect-cong)
    fix x show (a ∼ x) = (b ∼ x)
    proof
      from eqv have b ∼ a ..
      also assume ... ∼ x
      finally show b ∼ x .
    next
      note eqv
      also assume b ∼ x
      finally show a ∼ x .
    qed
  qed
  thus ⌊a⌋ = ⌊b⌋ by (simp only: class-def)
qed
```

9.2.3 Picking representing elements

We define the operation *pick* for selecting representing elements from equivalence classes, which are always inhabited due to reflexivity of the underlying relation of "∼". All representatives are equivalent; the specific one chosen by *pick* is left unspecified due to Hilbert's choice operator (cf. §8.5).

```
constdefs
  pick :: 'a::equiv quot ⇒ 'a
  pick A ≡ SOME a. A = ⌊a⌋

theorem pick-equiv [intro]: pick ⌊a⌋ ∼ a
proof (unfold pick-def)
  show (SOME x. ⌊a⌋ = ⌊x⌋) ∼ a
  proof (rule someI2)
    show ⌊a⌋ = ⌊a⌋ ..
    fix x assume ⌊a⌋ = ⌊x⌋
    hence a ∼ x ..
    thus x ∼ a ..
  qed
qed

theorem pick-inverse [intro]: ⌊pick A⌋ = A
proof (cases A)
  fix a assume a: A = ⌊a⌋
  hence pick A ∼ a by (simp only: pick-equiv)
  hence ⌊pick A⌋ = ⌊a⌋ ..
  with a show ?thesis by simp
```

qed

The following rules support canonical function definitions on quotient types (with ≤ 2 arguments). The general version covers conditional definitions; the simpler unconditional formulation is already sufficient for most applications.

theorem *quot-cond-function*:
$(\bigwedge X\ Y.\ P\ X\ Y \Longrightarrow f\ X\ Y \equiv g\ (pick\ X)\ (pick\ Y)) \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'.\ \lfloor x \rfloor = \lfloor x' \rfloor \Longrightarrow \lfloor y \rfloor = \lfloor y' \rfloor$
$\Longrightarrow P\ \lfloor x \rfloor\ \lfloor y \rfloor \Longrightarrow P\ \lfloor x' \rfloor\ \lfloor y' \rfloor \Longrightarrow g\ x\ y = g\ x'\ y') \Longrightarrow$
$P\ \lfloor a \rfloor\ \lfloor b \rfloor \Longrightarrow f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
(**is** *PROP ?def* $\Longrightarrow$ *PROP ?cong* $\Longrightarrow$ - $\Longrightarrow$ -)
proof −
assume *cong*: *PROP ?cong*
assume *PROP ?def* **and** $P\ \lfloor a \rfloor\ \lfloor b \rfloor$
hence $f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ (pick\ \lfloor a \rfloor)\ (pick\ \lfloor b \rfloor)$ **by** (*simp only*:)
also have $g\ (pick\ \lfloor a \rfloor)\ (pick\ \lfloor b \rfloor) = g\ a\ b$
proof (*rule cong*)
show $\lfloor pick\ \lfloor a \rfloor \rfloor = \lfloor a \rfloor$ **..**
moreover
show $\lfloor pick\ \lfloor b \rfloor \rfloor = \lfloor b \rfloor$ **..**
moreover
show $P\ \lfloor a \rfloor\ \lfloor b \rfloor$ **.**
ultimately
show $P\ \lfloor pick\ \lfloor a \rfloor \rfloor\ \lfloor pick\ \lfloor b \rfloor \rfloor$ **by** (*simp only*:)
qed
finally show *?thesis* **.**
qed

theorem *quot-function*:
$(\bigwedge X\ Y.\ f\ X\ Y \equiv g\ (pick\ X)\ (pick\ Y)) \Longrightarrow$
$(\bigwedge x\ x'\ y\ y'.\ \lfloor x \rfloor = \lfloor x' \rfloor \Longrightarrow \lfloor y \rfloor = \lfloor y' \rfloor \Longrightarrow g\ x\ y = g\ x'\ y') \Longrightarrow$
$f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
proof −
case *antecedent* **from** *this TrueI*
show *?thesis* **by** (*rule quot-cond-function*)
qed

end

9.3 Rational numbers

theory *Rational-Numbers* = *Quotient* + *Ring-and-Field*:[1]

The field of rational numbers is represented in the canonical fashion: we start with concrete fractions over integers, define standard algebraic operations on

[1] Theory *Ring-and-Field* is imported from [Bauer *et al.*, 2001].

fractions, and establish the corresponding congruence properties. The resulting structure is then abstracted by a quotient type construction.

9.3.1 Fractions over integers

The type of fractions

Type *fraction* is represented by the set of pairs over integers, with numerator and denominator components, such that the latter is always non-zero.

typedef *fraction* = $\{(a, b) :: int \times int \mid a\ b.\ b \neq 0\}$
proof
  show $(0, \#1) \in ?fraction$ **by** *simp*
qed

constdefs
  $fract :: int \Rightarrow int \Rightarrow fraction$
  $fract\ a\ b \equiv Abs\text{-}fraction\ (a, b)$
  $num :: fraction \Rightarrow int$
  $num\ Q \equiv fst\ (Rep\text{-}fraction\ Q)$
  $den :: fraction \Rightarrow int$
  $den\ Q \equiv snd\ (Rep\text{-}fraction\ Q)$

We derive basic properties of the selector operations, as well as canonical representation rules.

lemma *fract-num* [*simp*]: $b \neq 0 \Longrightarrow num\ (fract\ a\ b) = a$
  by (*simp add*: *fract-def num-def fraction-def Abs-fraction-inverse*)

lemma *fract-den* [*simp*]: $b \neq 0 \Longrightarrow den\ (fract\ a\ b) = b$
  by (*simp add*: *fract-def den-def fraction-def Abs-fraction-inverse*)

lemma *fraction-cases* [*cases type*: *fraction*]:
  $(\bigwedge a\ b.\ Q = fract\ a\ b \Longrightarrow b \neq 0 \Longrightarrow C) \Longrightarrow C$
proof −
  assume *r*: $\bigwedge a\ b.\ Q = fract\ a\ b \Longrightarrow b \neq 0 \Longrightarrow C$
  obtain *a b* **where** $Q = fract\ a\ b$ **and** $b \neq 0$
    by (*cases Q*) (*auto simp add*: *fract-def fraction-def*)
  thus *C* **by** (*rule r*)
qed

lemma *fraction-induct* [*induct type*: *fraction*]:
    $(\bigwedge a\ b.\ b \neq 0 \Longrightarrow P\ (fract\ a\ b)) \Longrightarrow P\ Q$
  by (*cases Q*) *simp*

Equivalence of fractions

We instantiate the generic theory of quotient types (cf. §9.2) by defining the "$\sim$" relation for fractions appropriately. The properties of equivalence relations

are proven as well, turning type *fraction* into an instance of the *equiv* class.

```
instance fraction :: eqv ..

defs (overloaded)
  equiv-fraction-def: Q ∼ R ≡ num Q * den R = num R * den Q

lemma equiv-fraction-iff:
    b ≠ 0 ⟹ b' ≠ 0 ⟹ (fract a b ∼ fract a' b') = (a * b' = a' * b)
  by (simp add: equiv-fraction-def)

lemma equiv-fractionI [intro]:
    a * b' = a' * b ⟹ b ≠ 0 ⟹ b' ≠ 0 ⟹ fract a b ∼ fract a' b'
  by (insert equiv-fraction-iff) blast

lemma equiv-fractionD [dest]:
    fract a b ∼ fract a' b' ⟹ b ≠ 0 ⟹ b' ≠ 0 ⟹ a * b' = a' * b
  by (insert equiv-fraction-iff) blast

instance fraction :: equiv
proof
  fix Q R S :: fraction
  {
    show Q ∼ Q
    proof (induct Q)
      fix a b :: int
      assume b ≠ 0 and b ≠ 0
      with refl show fract a b ∼ fract a b ..
    qed
  next
    assume Q ∼ R and R ∼ S
    show Q ∼ S
    proof (insert prems, induct Q, induct R, induct S)
      fix a b a' b' a'' b'' :: int
      assume b: b ≠ 0 and b': b' ≠ 0 and b'': b'' ≠ 0
      assume fract a b ∼ fract a' b' hence eq1: a * b' = a' * b ..
      assume fract a' b' ∼ fract a'' b'' hence eq2: a' * b'' = a'' * b' ..
      have a * b'' = a'' * b
      proof cases
        assume a' = 0
        with b' eq1 eq2 have a = 0 ∧ a'' = 0 by auto
        thus ?thesis by simp
      next
        assume a': a' ≠ 0
        from eq1 eq2 have (a * b') * (a' * b'') = (a' * b) * (a'' * b') by simp
        hence (a * b'') * (a' * b') = (a'' * b) * (a' * b') by (simp only: zmult-ac)
        with a' b' show ?thesis by simp
      qed
      thus fract a b ∼ fract a'' b'' ..
    qed
```

```
  next
    show Q ∼ R ⟹ R ∼ Q
    proof (induct Q, induct R)
      fix a b a' b' :: int
      assume b: b ≠ 0 and b': b' ≠ 0
      assume fract a b ∼ fract a' b'
      hence a * b' = a' * b ..
      hence a' * b = a * b' ..
      thus fract a' b' ∼ fract a b ..
    qed
  }
qed

lemma eq-fraction-iff:
    b ≠ 0 ⟹ b' ≠ 0 ⟹ (⌊fract a b⌋ = ⌊fract a' b'⌋) = (a * b' = a' * b)
  by (simp add: equiv-fraction-iff quot-equality)

lemma eq-fractionI [intro]:
    a * b' = a' * b ⟹ b ≠ 0 ⟹ b' ≠ 0 ⟹ ⌊fract a b⌋ = ⌊fract a' b'⌋
  by (insert eq-fraction-iff) blast

lemma eq-fractionD [dest]:
    ⌊fract a b⌋ = ⌊fract a' b'⌋ ⟹ b ≠ 0 ⟹ b' ≠ 0 ⟹ a * b' = a' * b
  by (insert eq-fraction-iff) blast
```

Operations on fractions

We define basic arithmetic operations on fractions and demonstrate their "well-definedness", i.e. congruence with respect to the underlying equivalence relation.

As it happens, this rather "trivial" technical issue turns out to be the most tedious part of the whole construction of rational numbers. We are careful to pass only basic operations on fractions through the quotient construction (0, +, unary −, ∗, and unary *inverse*). Further derived operations will be introduced later on for rational numbers only, without referring to the representation.

```
instance fraction :: zero ..
instance fraction :: plus ..
instance fraction :: minus ..
instance fraction :: times ..
instance fraction :: inverse ..

defs (overloaded)
  zero-fraction-def: 0 ≡ fract 0 #1
  add-fraction-def: Q + R ≡
    fract (num Q * den R + num R * den Q) (den Q * den R)
  minus-fraction-def: −Q ≡ fract (−(num Q)) (den Q)
  mult-fraction-def: Q * R ≡ fract (num Q * num R) (den Q * den R)
  inverse-fraction-def: inverse Q ≡ fract (den Q) (num Q)
```

lemma *is-zero-fraction-iff*: $b \neq 0 \Longrightarrow (\lfloor fract\ a\ b \rfloor = \lfloor 0 \rfloor) = (a = 0)$
by (*simp add*: *zero-fraction-def eq-fraction-iff*)

theorem *add-fraction-cong*:
$\lfloor fract\ a\ b \rfloor = \lfloor fract\ a'\ b' \rfloor \Longrightarrow \lfloor fract\ c\ d \rfloor = \lfloor fract\ c'\ d' \rfloor$
$\Longrightarrow b \neq 0 \Longrightarrow b' \neq 0 \Longrightarrow d \neq 0 \Longrightarrow d' \neq 0$
$\Longrightarrow \lfloor fract\ a\ b + fract\ c\ d \rfloor = \lfloor fract\ a'\ b' + fract\ c'\ d' \rfloor$
proof −
assume *neq*: $b \neq 0$ $b' \neq 0$ $d \neq 0$ $d' \neq 0$
assume $\lfloor fract\ a\ b \rfloor = \lfloor fract\ a'\ b' \rfloor$ **hence** eq_1: $a * b' = a' * b$ **..**
assume $\lfloor fract\ c\ d \rfloor = \lfloor fract\ c'\ d' \rfloor$ **hence** eq_2: $c * d' = c' * d$ **..**
have $\lfloor fract\ (a * d + c * b)\ (b * d) \rfloor = \lfloor fract\ (a' * d' + c' * b')\ (b' * d') \rfloor$
proof
show $(a * d + c * b) * (b' * d') = (a' * d' + c' * b') * (b * d)$ (**is** *?lhs* = *?rhs*)
proof −
have *?lhs* $= (a * b') * (d * d') + (c * d') * (b * b')$
by (*simp add*: *int-distrib zmult-ac*)
also have ... $= (a' * b) * (d * d') + (c' * d) * (b * b')$
by (*simp only*: eq_1 eq_2)
also have ... = *?rhs*
by (*simp add*: *int-distrib zmult-ac*)
finally show *?thesis* **.**
qed
from *neq* **show** $b * d \neq 0$ **by** *simp*
from *neq* **show** $b' * d' \neq 0$ **by** *simp*
qed
with *neq* **show** *?thesis* **by** (*simp add*: *add-fraction-def*)
qed

theorem *minus-fraction-cong*:
$\lfloor fract\ a\ b \rfloor = \lfloor fract\ a'\ b' \rfloor \Longrightarrow b \neq 0 \Longrightarrow b' \neq 0$
$\Longrightarrow \lfloor -(fract\ a\ b) \rfloor = \lfloor -(fract\ a'\ b') \rfloor$
proof −
assume *neq*: $b \neq 0$ $b' \neq 0$
assume $\lfloor fract\ a\ b \rfloor = \lfloor fract\ a'\ b' \rfloor$
hence $a * b' = a' * b$ **..**
hence $-a * b' = -a' * b$ **by** *simp*
hence $\lfloor fract\ (-a)\ b \rfloor = \lfloor fract\ (-a')\ b' \rfloor$ **..**
with *neq* **show** *?thesis* **by** (*simp add*: *minus-fraction-def*)
qed

theorem *mult-fraction-cong*:
$\lfloor fract\ a\ b \rfloor = \lfloor fract\ a'\ b' \rfloor \Longrightarrow \lfloor fract\ c\ d \rfloor = \lfloor fract\ c'\ d' \rfloor$
$\Longrightarrow b \neq 0 \Longrightarrow b' \neq 0 \Longrightarrow d \neq 0 \Longrightarrow d' \neq 0$
$\Longrightarrow \lfloor fract\ a\ b * fract\ c\ d \rfloor = \lfloor fract\ a'\ b' * fract\ c'\ d' \rfloor$
proof −
assume *neq*: $b \neq 0$ $b' \neq 0$ $d \neq 0$ $d' \neq 0$
assume $\lfloor fract\ a\ b \rfloor = \lfloor fract\ a'\ b' \rfloor$ **hence** eq_1: $a * b' = a' * b$ **..**
assume $\lfloor fract\ c\ d \rfloor = \lfloor fract\ c'\ d' \rfloor$ **hence** eq_2: $c * d' = c' * d$ **..**
have $\lfloor fract\ (a * c)\ (b * d) \rfloor = \lfloor fract\ (a' * c')\ (b' * d') \rfloor$

```
  proof
    from eq1 eq2 have (a * b') * (c * d') = (a' * b) * (c' * d) by simp
    thus (a * c) * (b' * d') = (a' * c') * (b * d) by (simp add: zmult-ac)
    from neq show b * d ≠ 0 by simp
    from neq show b' * d' ≠ 0 by simp
  qed
  with neq show ⌊fract a b * fract c d⌋ = ⌊fract a' b' * fract c' d'⌋
    by (simp add: mult-fraction-def)
qed

theorem inverse-fraction-cong:
  ⌊fract a b⌋ = ⌊fract a' b'⌋ ⟹ ⌊fract a b⌋ ≠ ⌊0⌋ ⟹ ⌊fract a' b'⌋ ≠ ⌊0⌋
    ⟹ b ≠ 0 ⟹ b' ≠ 0
    ⟹ ⌊inverse (fract a b)⌋ = ⌊inverse (fract a' b')⌋
proof −
  assume neq: b ≠ 0  b' ≠ 0
  assume ⌊fract a b⌋ ≠ ⌊0⌋ and ⌊fract a' b'⌋ ≠ ⌊0⌋
  with neq obtain a ≠ 0 and a' ≠ 0 by (simp add: is-zero-fraction-iff)
  assume ⌊fract a b⌋ = ⌊fract a' b'⌋
  hence a * b' = a' * b ..
  hence b * a' = b' * a by (simp only: zmult-ac)
  hence ⌊fract b a⌋ = ⌊fract b' a'⌋ ..
  with neq show ?thesis by (simp add: inverse-fraction-def)
qed
```

9.3.2 Rational numbers

The type of rational numbers

The type *rat* is represented as an abstraction of the universal set of quotient elements over fractions, wrt. the equivalence relation introduced before. We also provide abstract versions of the *pick* and ⌊-⌋ operations from quotients, called *fraction-of* and *rat-of*, respectively.

```
typedef (Rat)
  rat = UNIV :: fraction quot set ..

lemma RatI [intro, simp]: Q ∈ Rat
  by (simp add: Rat-def)

constdefs
  fraction-of :: rat ⇒ fraction
  fraction-of q ≡ pick (Rep-Rat q)
  rat-of :: fraction ⇒ rat
  rat-of Q ≡ Abs-Rat ⌊Q⌋

theorem rat-of-equality [iff?]: (rat-of Q = rat-of Q') = (⌊Q⌋ = ⌊Q'⌋)
  by (simp add: rat-of-def Abs-Rat-inject)
```

lemma *rat-of*: $\lfloor Q \rfloor = \lfloor Q' \rfloor \Longrightarrow$ *rat-of* $Q =$ *rat-of* Q' **..**

The canonical representation of rational numbers is by "fractional expressions" of the form $\langle a,\ b\rangle$. Subsequently, we also establish common equality rules and eliminations.

constdefs
  Fract :: *int* $\Rightarrow$ *int* $\Rightarrow$ *rat* (($\langle$-,/ -$\rangle$))
  $\langle a,\ b\rangle \equiv$ *rat-of* (*fract a b*)

theorem *Fract-inverse*: $\lfloor$*fraction-of* $\langle a,\ b\rangle\rfloor = \lfloor$*fract a b*$\rfloor$
  by (*simp add*: *fraction-of-def rat-of-def Fract-def Abs-Rat-inverse pick-inverse*)

theorem *Fract-equality* [*iff?*]: $(\langle a,\ b\rangle = \langle c,\ d\rangle) = (\lfloor$*fract a b*$\rfloor = \lfloor$*fract c d*$\rfloor)$
  by (*simp add*: *Fract-def rat-of-equality*)

theorem *eq-rat*: $b \neq 0 \Longrightarrow d \neq 0 \Longrightarrow (\langle a,\ b\rangle = \langle c,\ d\rangle) = (a * d = c * b)$
  by (*simp add*: *Fract-equality eq-fraction-iff*)

theorem *Rat-cases* [*cases type*: *rat*]:
  $(\bigwedge a\ b.\ q = \langle a,\ b\rangle \Longrightarrow b \neq 0 \Longrightarrow C) \Longrightarrow C$
proof −
  assume *r*: $\bigwedge a\ b.\ q = \langle a,\ b\rangle \Longrightarrow b \neq 0 \Longrightarrow C$
  obtain *x* **where** $q =$ *Abs-Rat x* **by** (*cases q*)
  moreover obtain *Q* **where** $x = \lfloor Q \rfloor$ **by** (*cases x*)
  moreover obtain *a b* **where** $Q =$ *fract a b* **and** $b \neq 0$ **by** (*cases Q*)
  ultimately have $q = \langle a,\ b\rangle$ **by** (*simp only*: *Fract-def rat-of-def*)
  thus *?thesis* **by** (*rule r*)
qed

theorem *Rat-induct* [*induct type*: *rat*]:
    $(\bigwedge a\ b.\ b \neq 0 \Longrightarrow P\ \langle a,\ b\rangle) \Longrightarrow P\ q$
  by (*cases q*) *simp*

Canonical function definitions

The generic definitional principle on quotient types (§9.2.3) is now transferred to rational numbers as follows. (The full conditional version will be only required for *inverse*; otherwise the simple unconditional formulation is sufficient.)

theorem *rat-cond-function*:
  $(\bigwedge q\ r.\ P\ \lfloor$*fraction-of q*$\rfloor\ \lfloor$*fraction-of r*$\rfloor \Longrightarrow$
    $f\ q\ r \equiv g$ (*fraction-of q*) (*fraction-of r*)) $\Longrightarrow$
   $(\bigwedge a\ b\ a'\ b'\ c\ d\ c'\ d'.$
    $\lfloor$*fract a b*$\rfloor = \lfloor$*fract a' b'*$\rfloor \Longrightarrow \lfloor$*fract c d*$\rfloor = \lfloor$*fract c' d'*$\rfloor \Longrightarrow$
    $P\ \lfloor$*fract a b*$\rfloor\ \lfloor$*fract c d*$\rfloor \Longrightarrow P\ \lfloor$*fract a' b'*$\rfloor\ \lfloor$*fract c' d'*$\rfloor \Longrightarrow$
    $b \neq 0 \Longrightarrow b' \neq 0 \Longrightarrow d \neq 0 \Longrightarrow d' \neq 0 \Longrightarrow$
    g (*fract a b*) (*fract c d*) $= g$ (*fract a' b'*) (*fract c' d'*)) $\Longrightarrow$
   $P\ \lfloor$*fract a b*$\rfloor\ \lfloor$*fract c d*$\rfloor \Longrightarrow f\ \langle a,\ b\rangle\ \langle c,\ d\rangle = g$ (*fract a b*) (*fract c d*)

```
  (is PROP ?def ⟹ PROP ?cong ⟹ ?P ⟹ -)
proof −
  assume eq: PROP ?def and cong: PROP ?cong and P: ?P
  have f (Abs-Rat ⌊fract a b⌋) (Abs-Rat ⌊fract c d⌋) = g (fract a b) (fract c d)
  proof (rule quot-cond-function)
    fix X Y assume P X Y
    with eq show f (Abs-Rat X) (Abs-Rat Y) ≡ g (pick X) (pick Y)
      by (simp add: fraction-of-def pick-inverse Abs-Rat-inverse)
  next
    fix Q Q′ R R′ :: fraction
    show ⌊Q⌋ = ⌊Q′⌋ ⟹ ⌊R⌋ = ⌊R′⌋ ⟹
        P ⌊Q⌋ ⌊R⌋ ⟹ P ⌊Q′⌋ ⌊R′⌋ ⟹ g Q R = g Q′ R′
      by (induct Q, induct Q′, induct R, induct R′) (rule cong)
  qed
  thus ?thesis by (unfold Fract-def rat-of-def)
qed

theorem rat-function:
  (⋀q r. f q r ≡ g (fraction-of q) (fraction-of r)) ⟹
    (⋀a b a′ b′ c d c′ d′.
      ⌊fract a b⌋ = ⌊fract a′ b′⌋ ⟹ ⌊fract c d⌋ = ⌊fract c′ d′⌋ ⟹
      b ≠ 0 ⟹ b′ ≠ 0 ⟹ d ≠ 0 ⟹ d′ ≠ 0 ⟹
      g (fract a b) (fract c d) = g (fract a′ b′) (fract c′ d′)) ⟹
    f ⟨a, b⟩ ⟨c, d⟩ = g (fract a b) (fract c d)
proof −
  case antecedent from this TrueI
  show ?thesis by (rule rat-cond-function)
qed
```

Standard operations on rational numbers

We are ready to provide the complete collection of arithmetic operations, according to the generic signature underlying the class *field* of [Bauer *et al.*, 2001].

```
instance rat :: zero ..
instance rat :: plus ..
instance rat :: minus ..
instance rat :: times ..
instance rat :: inverse ..
instance rat :: number ..

defs (overloaded)
  zero-rat-def: 0 ≡ rat-of 0
  number-of-rat-def: number-of b ≡ Fract (number-of b) #1
  add-rat-def: q + r ≡ rat-of (fraction-of q + fraction-of r)
  minus-rat-def: −q ≡ rat-of (−(fraction-of q))
  diff-rat-def: q − r ≡ q + (−(r::rat))
  mult-rat-def: q ∗ r ≡ rat-of (fraction-of q ∗ fraction-of r)
  inverse-rat-def: q ≠ 0 ⟹ inverse q ≡ rat-of (inverse (fraction-of q))
  divide-rat-def: r ≠ 0 ⟹ q / r ≡ q ∗ inverse (r::rat)
```

theorem *zero-rat*: $0 = \langle 0, \#1 \rangle$
by (*simp add*: *zero-rat-def zero-fraction-def rat-of-def Fract-def*)

theorem *add-rat*: $b \neq 0 \Longrightarrow d \neq 0 \Longrightarrow \langle a, b \rangle + \langle c, d \rangle = \langle a * d + c * b, b * d \rangle$
proof −
have $\langle a, b \rangle + \langle c, d \rangle = rat\text{-}of\ (fract\ a\ b + fract\ c\ d)$
by (*rule rat-function*, *rule add-rat-def*, *rule rat-of*, *rule add-fraction-cong*)
also
assume $b \neq 0 \quad d \neq 0$
hence $fract\ a\ b + fract\ c\ d = fract\ (a * d + c * b)\ (b * d)$
by (*simp add*: *add-fraction-def*)
finally show *?thesis* **by** (*unfold Fract-def*)
qed

theorem *minus-rat*: $b \neq 0 \Longrightarrow -\langle a, b \rangle = \langle -a, b \rangle$
proof −
have $-\langle a, b \rangle = rat\text{-}of\ (-(fract\ a\ b))$
by (*rule rat-function*, *rule minus-rat-def*, *rule rat-of*, *rule minus-fraction-cong*)
also assume $b \neq 0$ **hence** $-(fract\ a\ b) = fract\ (-a)\ b$
by (*simp add*: *minus-fraction-def*)
finally show *?thesis* **by** (*unfold Fract-def*)
qed

theorem *diff-rat*: $b \neq 0 \Longrightarrow d \neq 0 \Longrightarrow \langle a, b \rangle - \langle c, d \rangle = \langle a * d - c * b, b * d \rangle$
by (*simp add*: *diff-rat-def add-rat minus-rat*)

theorem *mult-rat*: $b \neq 0 \Longrightarrow d \neq 0 \Longrightarrow \langle a, b \rangle * \langle c, d \rangle = \langle a * c, b * d \rangle$
proof −
have $\langle a, b \rangle * \langle c, d \rangle = rat\text{-}of\ (fract\ a\ b * fract\ c\ d)$
by (*rule rat-function*, *rule mult-rat-def*, *rule rat-of*, *rule mult-fraction-cong*)
also
assume $b \neq 0 \quad d \neq 0$
hence $fract\ a\ b * fract\ c\ d = fract\ (a * c)\ (b * d)$
by (*simp add*: *mult-fraction-def*)
finally show *?thesis* **by** (*unfold Fract-def*)
qed

theorem *inverse-rat*: $\langle a, b \rangle \neq 0 \Longrightarrow b \neq 0 \Longrightarrow inverse\ \langle a, b \rangle = \langle b, a \rangle$
proof −
assume *neq*: $b \neq 0$ **and** *nonzero*: $\langle a, b \rangle \neq 0$
hence $\lfloor fract\ a\ b \rfloor \neq \lfloor 0 \rfloor$
by (*simp add*: *zero-rat eq-rat is-zero-fraction-iff*)
with - *inverse-fraction-cong* [*THEN rat-of*]
have $inverse\ \langle a, b \rangle = rat\text{-}of\ (inverse\ (fract\ a\ b))$
proof (*rule rat-cond-function*)
fix q **assume** *cond*: $\lfloor fraction\text{-}of\ q \rfloor \neq \lfloor 0 \rfloor$
have $q \neq 0$
proof (*cases q*)
fix $a\ b$ **assume** $b \neq 0$ **and** $q = \langle a, b \rangle$

```
    from this cond show ?thesis
      by (simp add: Fract-inverse is-zero-fraction-iff zero-rat eq-rat)
  qed
  thus inverse q ≡ rat-of (inverse (fraction-of q))
    by (rule inverse-rat-def)
 qed
 also from neq nonzero have inverse (fract a b) = fract b a
  by (simp add: inverse-fraction-def)
 finally show ?thesis by (unfold Fract-def)
qed

theorem divide-rat:
  ⟨c, d⟩ ≠ 0 ⟹ b ≠ 0 ⟹ d ≠ 0 ⟹ ⟨a, b⟩ / ⟨c, d⟩ = ⟨a * d, b * c⟩
proof −
 assume neq: b ≠ 0  d ≠ 0 and nonzero: ⟨c, d⟩ ≠ 0
 hence c ≠ 0 by (simp add: zero-rat eq-rat)
 with neq nonzero show ?thesis
  by (simp add: divide-rat-def inverse-rat mult-rat)
qed
```

The field of rational numbers

The final instantiation of type *rat* as a *field* is now imminent. This would make any infrastructure offered for general fields available for rational numbers as well (such as specific rules and proof tools).

```
instance rat :: field
proof
 fix q r s :: rat
 show (q + r) + s = q + (r + s)
  by (induct q, induct r, induct s) (simp add: add-rat zadd-ac zmult-ac int-distrib)
 show q + r = r + q
  by (induct q, induct r) (simp add: add-rat zadd-ac zmult-ac)
 show 0 + q = q
  by (induct q) (simp add: zero-rat add-rat)
 show (−q) + q = 0
  by (induct q) (simp add: zero-rat minus-rat add-rat eq-rat)
 show q − r = q + (−r)
  by (induct q, induct r) (simp add: add-rat minus-rat diff-rat)
 show (q * r) * s = q * (r * s)
  by (induct q, induct r, induct s) (simp add: mult-rat zmult-ac)
 show q * r = r * q
  by (induct q, induct r) (simp add: mult-rat zmult-ac)
 show #1 * q = q
  by (induct q) (simp add: number-of-rat-def mult-rat)
 show (q + r) * s = q * s + r * s
  by (induct q, induct r, induct s) (simp add: add-rat mult-rat eq-rat int-distrib)
 show q ≠ 0 ⟹ inverse q * q = #1
  by (induct q) (simp add: inverse-rat mult-rat number-of-rat-def zero-rat eq-rat)
 show r ≠ 0 ⟹ q / r = q * inverse r
```

by (*induct q*, *induct r*) (*simp add*: *mult-rat divide-rat inverse-rat zero-rat eq-rat*)
qed

end

9.4 Discussion

9.4.1 Isar techniques

Unusual calculations

The proof of theorem *quot-cond-function* in §9.2.3 exhibits an interesting example of a slightly unusual calculational sequence (cf. chapter 6). The most basic calculations merely consist of several **have** facts composed via **also** and **finally**, here we encounter a number of **show** statements linked via **moreover** and **ultimately**.

```
have g (pick ⌊a⌋) (pick ⌊b⌋) = g a b
proof (rule cong)
  show ⌊pick ⌊a⌋⌋ = ⌊a⌋ ..
  moreover
  show ⌊pick ⌊b⌋⌋ = ⌊b⌋ ..
  moreover
  show P ⌊a⌋ ⌊b⌋ .
  ultimately
  show P ⌊pick ⌊a⌋⌋ ⌊pick ⌊b⌋⌋ by (simp only:)
qed
```

In the above proof body we need to establish 4 sub-problems, where the first 3 contribute to the last one. Thus we have essentially two overlapping threads of reasoning, which have been connected via calculational proof commands. Recall that Isar's calculational elements are independent of the particular way that intermediate facts are produced, **show** works just the same as plain **have** (as far as the local result is concerned).

In the subsequent version of the proof, the basic flow of information is made more explicit by explicit labeling of facts.

```
have g (pick ⌊a⌋) (pick ⌊b⌋) = g a b
proof (rule cong)
  show 1: ⌊pick ⌊a⌋⌋ = ⌊a⌋ ..
  show 2: ⌊pick ⌊b⌋⌋ = ⌊b⌋ ..
  show 3: P ⌊a⌋ ⌊b⌋ .
  from 1 2 3 show P ⌊pick ⌊a⌋⌋ ⌊pick ⌊b⌋⌋ by (simp only:)
qed
```

While this form is probably more lucid wrt. the technical details, it is slightly more awkward due to excessive naming of statements. The situation is not that

bad after all, since we may get rid of the last label by the standard technique of chaining via **with** instead of **from** (cf. §4.2.4).

```
have g (pick ⌊a⌋) (pick ⌊b⌋) = g a b
proof (rule cong)
  show 1: ⌊pick ⌊a⌋⌋ = ⌊a⌋ ..
  show 2: ⌊pick ⌊b⌋⌋ = ⌊b⌋ ..
  show P ⌊a⌋ ⌊b⌋ .
  with 1 2 show P ⌊pick ⌊a⌋⌋ ⌊pick ⌊b⌋⌋ by (simp only:)
qed
```

Apparently, this works out reasonably well, because the original calculation has been short and did not apply any intermediate rules yet, deferring the equational composition of the ultimate result to the simplifier method in the very last step (cf. §6.4.3). Beginning users of Isar might prefer this plain formulation above over the neat calculational arrangement given before.

The following alternative formulation proceeds by individual substitution steps in the text, replacing the simplifier method. This requires some rearrangement of the order of sub-problems; we also swap the two equational facts in the calculational sequence (the use of *symmetric* below causes the local result to be swapped, while the version exported into the enclosing goal context is *not* affected).

```
have g (pick ⌊a⌋) (pick ⌊b⌋) = g a b
proof (rule cong)
  show P ⌊a⌋ ⌊b⌋ .
  also show [symmetric]: ⌊pick ⌊a⌋⌋ = ⌊a⌋ ..
  also show [symmetric]: ⌊pick ⌊b⌋⌋ = ⌊b⌋ ..
  finally show P ⌊pick ⌊a⌋⌋ ⌊pick ⌊b⌋⌋ .
qed
```

Thus we achieve an even more compact representation of the two intertwined threads of reasoning, which is not so easily replaced by plain forward-chaining anymore. On the other hand, this form is probably overly smart and might confuse readers unnecessarily.

Side conditions and proven assumptions

The present theory of fractions inherently involves numerous side-conditions about denominators being non-zero. This essentially amounts to subtyping, which we would like to keep in the proof texts as implicit as possible.

PVS [Owre *et al.*, 1996] treats such additional constraints mostly automatic by virtue of its integrated system of "predicate subtypes", which provides a reflection of logical statements within the type system. Note that PVS is essentially based on untyped set-theory, augmented by specific infrastructure to treat this particular kind of subtyping mostly behind the scenes. Nevertheless, PVS is usually marketed as another version of Higher-Order Logic.

As Isabelle/HOL is based on *simply-typed* set-theory according to the original tradition of HOL (e.g. [Gordon, 2000]), we need to take care of additional constraints directly within the logic. As we shall see, this works out reasonably well in Isar, without producing excessive formal noise in the text.

Throughout the theory we have followed the discipline to put side-conditions just before the main conclusion of a rule, after any number of main premises that tend to be filled-in by explicit forward-chaining of previous facts. A typical statement is that of theorem *inverse-fraction-cong* (cf. §9.3.1).

theorem *inverse-fraction-cong*:
$\lfloor fract\ a\ b \rfloor = \lfloor fract\ a'\ b' \rfloor \Longrightarrow \lfloor fract\ a\ b \rfloor \neq \lfloor 0 \rfloor \Longrightarrow \lfloor fract\ a'\ b' \rfloor \neq \lfloor 0 \rfloor$
$\Longrightarrow b \neq 0 \Longrightarrow b' \neq 0$
$\Longrightarrow \lfloor inverse\ (fract\ a\ b) \rfloor = \lfloor inverse\ (fract\ a'\ b') \rfloor$

Using such a rule in structured proof leaves any marginal side-conditions as open sub-problems to be solved at the very end. In most cases, this would just be done by assumption, referring implicitly to the current context. Sometimes such auxiliary facts emerge indirectly from previous ones. While plain **have** would require the result to be referenced explicitly, we may use a degenerate form of **obtain** (cf. §5.3) in order to acquire "proven assumptions" that are ready for implicit use later on.

The very proof of *inverse-fraction-cong* illustrates the use of both immediate and proven assumptions to handle side-conditions. In the version given below we have modified the text to point out implicit applications of standard rules and assumptions directly.

proof −
assume b: $b \neq 0$ **and** b': $b' \neq 0$ **note** $neq = b\ b'$
assume $\lfloor fract\ a\ b \rfloor \neq \lfloor 0 \rfloor$ **and** $\lfloor fract\ a'\ b' \rfloor \neq \lfloor 0 \rfloor$
with neq **obtain** a: $a \neq 0$ **and** a': $a' \neq 0$ **by** (*simp add*: *is-zero-fraction-iff*)
— acquire proven assumptions
assume $\lfloor fract\ a\ b \rfloor = \lfloor fract\ a'\ b' \rfloor$
hence $a * b' = a' * b$
by (*rule eq-fractionD*) (*rule b*, *rule b'*)
— apply standard rule, solving side-conditions by assumption
hence $b * a' = b' * a$ **by** (*simp only*: *zmult-ac*)
hence $\lfloor fract\ b\ a \rfloor = \lfloor fract\ b'\ a' \rfloor$
by (*rule eq-fractionI*) (*rule a*, *rule a'*)
— apply standard rule, solving side-conditions by assumption
with neq **show** *?thesis* **by** (*simp add*: *inverse-fraction-def*)
qed

Side-conditions of the same kind need to be treated over and over again in the present application. Nevertheless, the formal detail required here is still at a bearable level, due to the virtue of Isar proof processing to consider a problem as solved up to immediate assumptions (cf. §3.2.3). This discipline is easily

achieved due to the inherent structure of Isar proof texts, with local problems being clearly delimited.

In contrast, unstructured proof scripts would usually require even "trivial" assumption steps to be given explicitly by the user. This results in more cumbersome treatment of the numerous side-conditions encountered in the present example. So users of tactic scripts would probably demand separate proof tools for such specific situations.

Representation proofs

Our construction of the type of rational numbers involves a number of representation proofs, in the sense that existing elements are considered as an image of certain functions (with the domain expressed by additional conditions). The **obtain** language element is particularly well-suited for this kind of reasoning.

The theory development eventually arrives at the canonical representation via fractional expressions; cf. theorems *Rat-cases* and *Rat-induct* in §9.3.2. The proof of theorem *Rat-cases* itself is particularly interesting, since it involves the complete hierarchy of individual representations stemming from several HOL **typedef** specifications that contribute to type *rat*.

Consequently, the proof has used a number of **obtain** statements (cf. §5.3), composed by calculational elements (cf. chapter 6). Note that the 3rd occurrence of **obtain** below plays a second role in introducing a proven assumption $b \neq 0$ for the side-condition to be covered implicitly in the final step.

theorem *Rat-cases* [*cases type*: *rat*]:
  $(\bigwedge a\ b.\ q = \langle a,\ b\rangle \Longrightarrow b \neq 0 \Longrightarrow C) \Longrightarrow C$
proof −
  assume *r*: $\bigwedge a\ b.\ q = \langle a,\ b\rangle \Longrightarrow b \neq 0 \Longrightarrow C$
  obtain *x* **where** $q = \mathit{Abs\text{-}Rat}\ x$ **by** (*cases q*)
  moreover obtain *Q* **where** $x = \lfloor Q \rfloor$ **by** (*cases x*)
  moreover obtain *a b* **where** $Q = \mathit{fract}\ a\ b$ **and** $b \neq 0$ **by** (*cases Q*)
  ultimately have $q = \langle a,\ b\rangle$ **by** (*simp only*: *Fract-def rat-of-def*)
  thus *?thesis* **by** (*rule r*)
qed

Isar calculations work with any kind of facts; from this perspective there is nothing special about **obtain**, as opposed to **have**, **note**, **assume** etc. (chapter 6). In contrast, the builtin concept of iterated equalities in Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] is quite picky about the exact format required here. In particular, Mizar's `consider` and `given` (which are analogous to Isar's **obtain**) may not be used together with complex proof patterns involving linking of facts or iterated equalities.

Due to Isar's flexible approach to calculational reasoning, we have been able to traverse the nested representations of type *rat* adequately (following the structure as an abstracted version of *fraction quot*, cf. §9.3.2). Later on, results about

abstract rational numbers would directly use the derived scheme of *Rat-cases*, or its variant form *Rat-induct* presented in induction rule format.

For example, the proof of the field properties of type *rat* (§9.3.2) involves a number of universal statements about rational numbers. Using the form of (degenerate) induction that has been derived for type *rat*, we reduce the problem to fractional expressions; the corresponding algebraic laws on these canonical representations have already been established before.

fix $q\ r\ s$ :: *rat*
show $(q + r) + s = q + (r + s)$
  by (*induct q*, *induct r*, *induct s*) (*simp add*: *add-rat zadd-ac zmult-ac int-distrib*)

Here we give an enlarged version of this kind of reasoning, where the reduced universal statements are made explicit in the text.

fix $q\ r\ s$ :: *rat*
show $(q + r) + s = q + (r + s)$
proof (*induct q*, *induct r*, *induct s*)
  fix $a\ b\ c\ d\ e\ f$ :: *int*
  assume $b \neq 0\ \ d \neq 0\ \ f \neq 0$
  thus $(\langle a, b\rangle + \langle c, d\rangle) + \langle e, f\rangle = \langle a, b\rangle + (\langle c, d\rangle + \langle e, f\rangle)$
    by (*simp add*: *add-rat zadd-ac zmult-ac int-distrib*)
qed

Note that in the instantiation proof of class *field* the conditional laws about the *inverse* and "/" operations depend on the ability of *induct* to handle non-atomic statements properly (cf. §5.4.5).

show $q \neq 0 \Longrightarrow \mathit{inverse}\ q * q = \#1$ — (q occurs in premise)
  by (*induct q*) (*simp add*: *inverse-rat mult-rat number-of-rat-def zero-rat eq-rat*)
show $r \neq 0 \Longrightarrow q\ /\ r = q * \mathit{inverse}\ r$ — (r occurs in premise)
  by (*induct q*, *induct r*) (*simp add*: *mult-rat divide-rat inverse-rat zero-rat eq-rat*)

Isar generally supports non-atomic statements in a uniform manner (cf. §5.2.5 and §5.4.5). Otherwise, we would have required the notorious detour via object-level connectives in order to be able to apply induction rules (cf. the discussion of induction in Isabelle proof scripts in [Nipkow and Paulson, 2001]). As the present form of induction is rather degenerate indeed, such additional overhead would have made our approach to representation proofs a less satisfactory one, with the additional formal noise dominating an essentially trivial proof scheme.

9.4.2 HOL techniques

The HOL logic (cf. chapter 7 and chapter 8) underlying the Isabelle/HOL working environment admits several distinctive techniques based on inherent features of the system that might occur slightly peculiar at first sight. Nevertheless, these may become quite useful in tackling a few recurrent issues of formal logic

applications in a simple manner. Here we focus on type abstraction and underspecification, as encountered in our formulation of rational numbers.

Abstract types versus type abstraction

Abstraction is certainly an important issue for any kind of formal specification environment. In general, specifications should be presented at an "adequate" level of abstraction wherever possible. First of all, overly concrete mathematical models may be crowded by irrelevant detail, which needs to be taken care of in formal proofs. Furthermore, some *accidental properties* of a specific representation may get exploited by users later on, although not intended by the writer of the specification. The latter effect is both philosophically unpleasing and may lead into practical problems due to lacking modularity of theories.

On the other hand, extremely abstract presentations of mathematical ideas may be quite far from what users would expect in the first place. A typical example of this issue are abstract categorical characterizations of "well-known" mathematical concepts. There is also some additional demand for logical tools and techniques for highly abstract presentations. Users versed in techniques of algebraic specification and model theory would probably request rather general theory mechanisms and module systems, based on powerful features of category theory, for example. These are not readily available in practically relevant logics such as HOL (cf. chapter 7).

The issue of abstract versus concrete specification techniques is occasionally linked to the *axiomatic* versus *definitional* method of exhibiting mathematical results. In principle, a strictly definitional approach would easily lead to unexpected facts holding in the resulting theory development. A common example of classical mathematics based on untyped set-theory is the definition of natural numbers according to John von Neumann: $0 = \{\}$, $1 = \{0\}$, $2 = \{0, 1\}$ etc. Most mathematicians would probably just choose to ignore accidental properties like $1 \in 2$ or $1 \subseteq 2$, while set-theory people would even make creative use of such features to develop further useful concepts, e.g. ordinal numbers.

The HOL tradition is strongly biased towards the definitional approach (cf. §7.1.2), using plain mathematical modeling as its key specification technique. It might appear at first sight that notions of "abstract data types", which are commonplace in algebraic specification methods, would be quite alien to HOL. Interestingly, HOL is able to reconcile the definitional and axiomatic methods by proper use of its inherent virtues, though.

The key observation is that HOL's **typedef** mechanism (cf. §7.1.2 and §8.6) involves an inherent abstraction stage: the only link between the existing representing set and the new type is via the fully abstract *rep* and *abs* functions. Thus **typedef** acts very much like type abstraction in higher-order programming languages, e.g. `abstype` in (old) ML.

For example, reconsider our present construction of rational numbers, based on

a concrete representation of fractions as pairs of integers (cf. §9.3.1). While this concrete mathematical model has enabled us to derive the intended properties of the abstract idea of rational numbers, we have not been able to exploit any accidental properties of the concrete representation in a meaningful manner.

In order to see this, assume a fully abstract axiomatic presentation of rational numbers within HOL. Clearly, this enables us to prove the canonical presentation of rational numbers in terms of "fractional expressions" (cf. *Rat-cases* in §9.3.2); from here we may work backwards towards the primitive representations of quotient elements and concrete fractions, essentially *deriving* the abstract characterization of the original **typedef** specifications as a theorem stating $\exists\, rep\ abs.\ type\text{-}definition\ rep\ abs\ \{\lfloor(a,\ b)\rfloor \mid a\ b.\ b \neq 0\}$ (cf. §8.6).

Note that the particular version of **typedef** in Isabelle/HOL includes another (conservative) stage to introduced concrete constants naming these bijections, which is not present in the original type definition primitive of the HOL formulation [Gordon and Melham, 1993] [Pitts, 1993].

We see that HOL indeed provides a two-way path from type abstraction of concrete mathematical structures back and forth to fully abstract types. It is unclear whether this particular view on HOL type definitions has been considered as an important issue by its original designers (cf. the historical account given in [Gordon, 2000]). Due to the way that HOL types are treated as a purely syntactic classification of objects, there is not much freedom left in providing a non-trivial definitional mechanism for types anyway. In particular, notions of convertibility (or equality) of types are absent in HOL; so any link between existing types and new ones need to be via abstract morphisms. There is no way to identify elements of different HOL types directly.

As far as users of HOL are concerned, the **typedef** primitive is widely considered too arcane to be used directly in applications. Ever since the important special case of inductive datatypes has become widely available in HOL implementations (cf. the overview given in [Berghofer and Wenzel, 1999]), users have generally preferred concrete mathematical models based on a general class of tree structures. Certainly, this often leads to specifications that are less abstract than necessary, e.g. using lists instead of finite multisets, or association lists (i.e. lists of pairs) instead of finitary functions (or general relations).

The present Isabelle/Isar example demonstrates that HOL type definitions may get used directly in applications, even in classical mathematics. Due to the high-level characterization of **typedef** via rules in *cases* or *induct* format (§7.1.2, §8.6), we have been able to keep the formal noise at a reasonable level; the generic **obtain** element (§5.3) has been a great help, too.

Partiality versus underspecification

Another recurrent issue in formal specification is that of "undefined" elements. The present example of rational numbers already exhibits the most prominent

instance of undefinedness, namely division by zero. People being introduced to elementary arithmetic are usually given rather mysterious explanations about the exact nature of $q \mathbin{/} 0$, telling that it is "not permitted" to divide by zero.

Even in established (semi-formal) mathematics, there are quite different traditions of treating "definedness" issues of expressions. Sometimes side-conditions like "provided that $q \mathbin{/} r$ is defined" are spelled out explicitly. Occasionally, one even encounters statements of the form "for $q \mathbin{/} r \in \mathbb{Q}$", based on the idea that $q \mathbin{/} r$ must not be a rational number in the "undefined" case of $r = 0$, otherwise one could conclude general facts like $0 * (q \mathbin{/} 0) = 0$. Note that a formal representation of the second idea may be achieved by "lifting" of basic types, i.e. by adjoining an explicit "error element" $\mathbb{Q} \cup \{\bot\}$, where $\bot \notin \mathbb{Q}$.

In contrast, formal HOL is strongly biased towards totality, in the sense that every syntactically well-formed type or term is always denoting something, although it might be genuinely "unspecified". Nevertheless, partial functions may be easily represented via lifting, e.g. using the type $'a \Rightarrow 'b\ option$ in Isabelle/HOL (cf. chapter 7); see [Müller and Slind, 1997] for further details on treating partiality in a total setting. The general disadvantage of modeling partiality explicitly is that it needs to be handled within formal proofs all the time. So operations that are "almost total" (like division on rational numbers) should better avoid this additional overhead. Consequently, we have preferred the plain type of total HOL functions in our theory, using $inverse :: rat \Rightarrow rat$ and $/ :: rat \Rightarrow rat \Rightarrow rat$.

There have been several attempts to reformulate the basic ideas of HOL with partiality in mind, e.g. in the "Lutins" logic underlying IMPS [Farmer *et al.*, 1993]. Additional builtin support for automated totality reasoning is required to turn the basic idea of first-class partiality into a practically useful environment.

The system of "predicate subtypes" of PVS [Owre *et al.*, 1996] may get used to model partial functions as well. Definedness reasoning would then be part of the builtin semi-automated treatment of "type-checking conditions" (TCCs).

Systems based on the tradition of dependent type theory (e.g. Coq [Barras *et al.*, 1999]) would usually treat the present case of partiality by including the non-zero property of divisors in the type of the division operator. While this is theoretically very clean, it demands additional efforts in practical applications. For example, definedness of divisors needs essentially be proven before being able to write down expressions involving division; cf. the experience reported in [Geuvers *et al.*, 2000].

According to the main-stream tradition of using HOL, the informal concept of "partial functions" is usually avoided altogether, just by inventing suitable results for "undefined" cases. For example, $q \mathbin{/} 0$ could be forced to yield 0. Thus we would also gain several useful algebraic properties to hold unconditionally, such as $\vdash inverse\ (inverse\ x) = x$ or $\vdash inverse\ (-x) = -(inverse\ x)$. Accidental properties like this often become quite useful for simplified treatment in specific proof tools, cf. the related discussion on the field of real numbers given

in [Harrison, 1996c].

Further established disciplines of treating "pseudo-partiality" in HOL involve Hilbert's choice operator (cf. §8.5) applied to the *empty* predicate. A similar technique uses a universal unspecified dummy elements, like *arbitrary* :: $'a$ in Isabelle/HOL [Nipkow *et al.*, 2001]. Speaking in terms of the standard model-theory of HOL [Pitts, 1993] such expressions denote fixed elements of a given type, but are completely unknown within the formal system. Nevertheless, nothing prevents the user to include such dummies in logical reasoning; basic facts like $\vdash$ *arbitrary* = *arbitrary* are certainly derivable within the logic. Thus we may again observe unwanted accidental results arising from uncontrolled reasoning with coinciding instances of such "arbitrary" expressions, which may have appeared as independent at first sight. A more robust solution would essentially require a separate copy of *arbitrary* for each occurrence in the specification text.

Longterm experience with "arbitrary" HOL expressions has shown that such tricks easily confuse uninitiated recipients. Including such features in a presentation given to non-experts of HOL one is apt to distract the audience from the main issues for quite some time.

As demonstrated by the present application of Isar, we may achieve a slightly cleaner treatment of "undefined" expressions in full accordance with established HOL traditions. Taking the term "undefined" literally, we simply exclude certain unwanted cases from the definition of a total function, achieving genuine underspecification rather than explicit partiality. This idea has been expressed natively via conditional definitions (of overloaded constants) in §9.3.2.

inverse-rat-def: $q \neq 0 \Longrightarrow$ *inverse* $q \equiv$ *rat-of* (*inverse* (*fraction-of* q))
divide-rat-def: $r \neq 0 \Longrightarrow q \mathbin{/} r \equiv q * \mathit{inverse}\ (r{::}\mathit{rat})$

There is nothing special about "partial" HOL definitions (cf. §7.1.2), conditional equations are a trivial instance of the notion of conservative extensions employed by the basic framework (cf. §2.3). Nevertheless, this form is rarely encountered in existing applications of HOL, despite being very useful.

By ruling out pathological cases from the very beginning, we may be sure that no accidental result involving specific treatment of *inverse* 0 and $q \mathbin{/} 0$. Nevertheless, HOL remains faithful to its initial totality approach, so we may derive instances of universal algebraic laws, such as $0 * (q \mathbin{/} 0) = 0$ (since $q \mathbin{/} 0$ is guaranteed to be a rational number by virtue of its syntactic type). On the other hand, *inverse* $(-x) = -(\mathit{inverse}\ x)$ does not hold unconditionally, since we cannot derive anything specific about $r \mathbin{/} q$ in the case of $q = 0$ that has been excluded from the definition.

9.4.3 Arithmetic proof tools

Our present construction of the rational numbers based on fractions over integers involves numerous instances of basic arithmetic reasoning.

The common technique to handle such incidents has been to tweak Isabelle's simplifier, in order to make such "obvious" local statements work out in a single step. Additional rewrite rules need to be specified whenever the (limited) builtin support for integers arithmetic did not suffice. For example, some typical *simp* method invocations taken from the instantiation proof of *rat* :: *field* (cf. §9.3.2).

(*simp add*: *add-rat zadd-ac zmult-ac int-distrib*)
(*simp add*: *add-rat zadd-ac zmult-ac*)
(*simp add*: *mult-rat zmult-ac*)
(*simp add*: *add-rat mult-rat eq-rat int-distrib*)

We see that "relevant" theorems about rational numbers (*add-rat*, *mult-rat* etc.) get mixed with basic arithmetic facts about integers (*zadd-ac*, *zmult-ac* etc.), which have been picked from the Isabelle/HOL library in an ad-hoc fashion.

This situation is slightly unsatisfactory, especially since there are many well-known (semi-)decision procedures for interesting classes of arithmetic problems in the literature. Isabelle/HOL happens to implement only very basic support for a restricted class of linear arithmetic on integers (without multiplication) [Nipkow *et al.*, 2001]. Other systems like Coq [Barras *et al.*, 1999], HOL [Gordon and Melham, 1993], and PVS [Owre *et al.*, 1996] offer their own collection of arithmetic procedures with quite varying coverage of specific problems. PVS is marketed as particularly strong in this respect; HOL-Light [Harrison, 1996a] implements full arithmetic procedures for integers (Presburger arithmetic) and the closed field of real numbers (according to Tarski).

In any case, users of interactive theorem proving systems routinely encounter rather frustrating situations where the builtin support for arithmetic fails, requiring particular cases to be proven by hand.

In principle, we have encountered the same situation in the present Isabelle/Isar application many times. Nevertheless, the resulting proof text turns out to be still quite acceptable. The lack of specific proof support did not affect the basic structure of our proofs, but has only required more detailed proof method specifications (especially of *simp* as indicated above), or prolonged a number of calculation chains by demanding more detailed steps to be given.

Essentially, our ability in Isar to decompose failed atomic proof steps into fine-grained arrangements of sub-problems (involving calculational reasoning) has compensated the lack of powerful arithmetic automated proof support to some degree. Apparently, high-level Isar proof elements have been able to magnify the strength of the underlying inference systems.

A similar effect may be experienced in Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999]. The builtin notion of "obvious inferences" [Rudnicki, 1987] is rather weak compared to existing proof tools of Isabelle [Paulson and Nipkow, 1994] or PVS [Owre *et al.*, 1996], neither is there is any specific support for arithmetic problems. Nevertheless, Mizar users have been able to develop a large body of classical mathematics [Mizar library], including plenty of standard issues about arithmetic of integers, rationals, reals, etc. A

frequently encountered technique is that of long chains of "iterated equalities", which are analogous to Isar's calculations (cf. chapter 6).

Apparently, the lack of decent proof tools did not prevent useful work to be done in Mizar, by virtue of adequate means to arrange the tedium of formal reasoning in a structured manner. The same observation should be valid for Isar as well. Of course, powerful tools for arithmetic problems would be useful to support even larger applications. In any case, the Isar framework is able to incorporate any specific proof methods that happen to become available as tactic implementations in the underlying inference engine (cf. §7.3).

Chapter 10

Example: Unix security

Unix is a simple but powerful operating system where everything is either a process or a file. Access to system resources works mainly via the file-system, including special files and devices, so Unix security issues are reflected directly within the file-system. We give a mathematical model of the main aspects of the Unix file-system including its security model, but ignoring processes. Within this formal model we discuss some aspects of Unix security, including a few odd effects caused by the general "worse-is-better" approach followed in Unix.

The resulting formal development demonstrates that Isabelle/Isar is sufficiently flexible to cover the typical abstract modeling and verification tasks encountered in computer-science applications of formal logic. So far this has been mainly the domain of interactive theorem proving systems using unstructured tactic languages.

10.1 Motivation

Over the last few years, tactical theorem proving systems like HOL [Gordon and Melham, 1993], Coq [Barras *et al.*, 1999], PVS [Owre *et al.*, 1996], and Isabelle [Paulson and Nipkow, 1994] have been successfully applied to sizable applications, especially of those of computer-science, involving abstract modeling and verification tasks. For example, the Isabelle/Bali project [Bali] [Oheimb, 2001] provides an extensive formalization of several aspects of the Java programming environment, covering the Java type system, operational semantics, and axiomatic semantics.

Applications of this kind heavily depend on a number of characteristic techniques for specification and formal proof, such as inductive definitions (of types, sets, relations), recursive function definitions, and corresponding proof principles by case-analysis and induction. Typically, the structures encountered here

are quite large, with many constructors or inductive cases, but not necessarily very deep with respect to the mathematical concepts involved.

The usual outcome of the work of many person-months (or even person-years) is then a body of theory definitions plus a large collection of proof scripts. According to widely accepted practice, the public presentation of such results involves extensive discussions of the definitional part (including particular design decisions in the formalization etc.), while actual formal proofs are mostly neglected. There is a strong tendency to report on the size and complexity of proof scripts, though, especially the degree of automation achieved by the collection of proof tools available in the preferred theorem proving environment.

For example, the Java formalization of [Oheimb, 2001] consists of 1900 lines of definitions and 5400 lines of tactic scripts. The definitions are included as an appendix to document the actual formal representation, while proof scripts are not given at all. Inspecting the real sources reveals that proof scripts follow the typical style of advanced tactical proving, with heavy use of proof programming techniques (operating on several sub-problems at the same time, or accommodating specific structures encountered in the formalization). [Oheimb, 2001] also covers a small synthetic example of concrete Java program verification; this part includes an account of the technical complexity of proof scripts, since it is an important parameter of the usability of the Java meta-theory in concrete applications. In contrast, the proofs of the meta-theory are not covered at all.

Marginal treatment of formal proofs is commonplace in contemporary applications of the class of interactive theorem proving systems considered here. Some systems even keep proof scripts apart from the specification parts of a theory by separate technical means. For example, PVS [Owre *et al.*, 1996] only holds the statements of theorems in the actual theory source, while proof scripts are managed separately (with some additional infrastructure for lemma dependencies and change management). Classic Isabelle [Paulson and Nipkow, 1994] keeps proof scripts in separate ML files, too, partly for purely technical reasons which have been overcome in Isabelle99. Degrading proof scripts to second class was nevertheless generally accepted among experienced Isabelle users as a natural order, despite causing a number of practical inconveniences (cf. §7.5.1).

In contrast, Coq [Barras *et al.*, 1999] keeps definitions and proof scripts within the same input source, reflecting the intrinsic virtue of type theory that both are actually the same internally, despite being composed by different technical means (immediate λ-terms versus incremental tactic applications). The original LCF/HOL tradition [Gordon, 2000] treats definitions and proofs uniformly as ML programs (mostly restricted to applications of certain standard functions).

The deeper cause for the general disregard of actual proofs in common applications of mainstream formal reasoning systems may be essentially twofold. On the one hand, existing tactical theorem proving systems do not provide any means to express proofs in a human-readable way in the first place. On the other hand, people who use (and develop) such systems often consider readable proofs as unimportant, being content to present formal accounts of their work

to the machine only. Certainly, these two aspects depend on each other; existing systems just happen to have evolved to support this particular proof-less mode of operation, which then happen to support a particular range of applications in a reasonable manner. This might also explain why there are less applications from traditional mathematics encountered in this area. Interestingly, the large body of standard analysis developed in [Harrison, 1996c] mainly serves as an auxiliary theory for program verification issues as well.

In contrast, the Mizar system [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999] has been able to support a large library of formalized mathematics [Mizar library] of approximately 50 MB of sources organized in 700 articles, which has been developed over 2 decades. This successful case of machine-checked proof developments made accessible for human consumption probably also draws from established practice of mainstream mathematics where real proofs are not given up lightly.

From the present Isar perspective, the key question to be raised here is whether the case for inclusion of actual formal proofs may be extended beyond pure logic (e.g. chapter 4, chapter 8) and classical mathematics (e.g. chapter 9) to computer-science applications involving abstract modeling and verification that are still mainly the domain of existing tactical systems. To this end, we shall present a non-trivial formal development *completely* with all definitions and formal proofs. In particular, proof texts may stand for themselves; we shall refrain from reporting accidental technical properties, such as the degree of automation achieved.

Another important question is whether exposing real proofs to the audience of computer-science applications makes any sense at all. Certainly, badly written proofs are better excluded from general public coverage. On the other hand, lucid formal expositions may greatly benefit from being more widely accepted as a viable account of the work achieved here, as well as help others to conduct similar or even more advanced applications of formal logic. There is no indication that this particular area of applied logic should be excluded from the free flow of ideas and techniques. If we really want to prevent relevant formal developments to be treated like black-boxes that are disclosed from public view we do need suitable means to make the resulting developments accessible to human consumption.

Human-readable proofs have already been encountered in computer-science applications before. [Mizar library] includes a number articles that qualify to belong to this domain, such as the meta-theory of classical first-order logic (articles #93, #97, #199, #253, #375), Petri nets (articles #137, #261, #262, #263, #298, #330), and over 30 articles about a "small computer model" (SCM) with both theoretical results and several concrete example programs (including correctness proofs). Certainly, these examples are only very few compared to the remaining body of classical mathematics formalized in Mizar. Furthermore, Mizar has been designed as a "closed" system for the particular domain of standard mathematical applications, there is no way to introduce new definitional

mechanisms or specialized proof tools; thus verification tasks turn out as slightly cumbersome, performing inductive constructions manually within primitive set-theory, and conducting routine proofs over many pages in rather small technical reasoning steps, for example.

DECLARE [Syme, 1997a] [Syme, 1998] [Syme, 1999] has been designed as a specific system for "declarative theorem proving" about syntax and semantics of formal languages in the first place. The system has been evaluated by an example development about Java type soundness [Syme, 1997b] [Syme, 1998].

The key difference to Isabelle/Isar is that the latter has not been *made* to work for a fixed application domain, but *grown* to cover a broad range, following the tradition of "generic theorem proving" of the Isabelle framework [Paulson, 1989] [Paulson, 1990] [Paulson and Nipkow, 1994], extending it to the upper levels of human-readable proof texts. Thus by providing a number of general principles that may be combined in many ways, we are able to cover the present range of applications of formal-reasoning systems from end-to-end, including primitive logic just as well as advanced modeling and verification problems encountered in computer-science. Certainly, the Isabelle/Isar system is open for even further areas to be explored by creative users.

The subsequent development of some aspects of Unix file-system security essentially demands all of the advanced Isar techniques covered in chapter 5 and chapter 6. Nevertheless, we need not really stretch the system very far, even larger applications could be performed quite easily. E.g. see Isabelle/MicroJava [Klein *et al.*, 2001], which consists of several structured proof texts, although it includes numerous old theories using unreadable proof scripts.

10.2 Introduction

10.2.1 The Unix philosophy

Over the last three decades the Unix community has collected a certain amount of folklore wisdom on building systems that actually work, see [Unix Heritage Society] and [PDP Unix Preservation Society] for further background information. The following account of the philosophical principles behind the Unix way of software and systems engineering have appeared on http://slashdot.com (25-March-2000).

```
The UNIX Philosophy (Score:2, Insightful)
by yebb on Saturday March 25, @11:06AM EST (#69)
(User Info)

The philosophy is a result of more than twenty years of software
development and has grown from the UNIX community instead of being
enforced upon it. It is a defacto-style of software development. The
nine major tenets of the UNIX Philosophy are:

  1. small is beautiful
```

```
  2. make each program do one thing well
  3. build a prototype as soon as possible
  4. choose portability over efficiency
  5. store numerical data in flat files
  6. use software leverage to your advantage
  7. use shell scripts to increase leverage and portability
  8. avoid captive user interfaces
  9. make every program a filter

The Ten Lesser Tenets

  1. allow the user to tailor the environment
  2. make operating system kernels small and lightweight
  3. use lower case and keep it short
  4. save trees
  5. silence is golden
  6. think parallel
  7. the sum of the parts if greater than the whole
  8. look for the ninety percent solution
  9. worse is better
 10. think hierarchically
```

The "worse-is-better" approach quoted above is particularly interesting. It basically means that *relevant* concepts have to be implemented in the right way, while *irrelevant* issues are simply ignored in order to avoid unnecessary complication of the design and implementation. Certainly, the overall quality of the resulting system heavily depends on the virtue of distinction between the two categories of "relevant" and "irrelevant".

10.2.2 Unix security

The main entities of a Unix system are *files* and *processes* (e.g. [Tanenbaum, 1992]). Files subsume any persistent "static" entity managed by the system, ranging from plain files and directories, to special ones such as device nodes, pipes, sockets etc. On the other hand, processes are "dynamic" entities that may perform certain operations while being run by the system.

The security model of classic Unix systems is centered around the file system. The operations permitted by a process that is run by a certain user are determined from information stored within the file system. This includes any kind of access control, e.g. read/write access to some plain file, or read-only access to a certain global device node etc. Thus proper arrangement of the main Unix file-system is very critical for overall security. (Incidently, this is the deeper reason why the operation of mounting new volumes into the existing file space is usually restricted to the super-user.)

Generally speaking, the Unix security model is a very simplistic one. The original designers did not have maximum security in mind, but wanted to get a decent system working for typical multi-user environments. Contemporary Unix implementations still follow the basic security model of the original versions from the early 1970's [Ritchie and Thompson, 1974]. Even back then there would have

been better approaches available, albeit with more complexity involved both for implementers and users.

On the other hand, even in the 2000's many computer systems are run with little or no file-system security at all, even though virtually any system is exposed to the net. Even "personal" computer systems have long left the comfortable home environment and entered the wilderness of the open net sphere.

This treatment of file-system security is a typical example of the "worse-is-better" principle introduced above. The simplistic security model of Unix got widely accepted within a large user community, while the more innovative (and cumbersome) ones are only used very reluctantly and even tend to be disabled by default in order to avoid confusion of beginners.

10.2.3 Odd effects

Simplistic systems usually work very well in typical situations, but tend to exhibit some odd features in atypical ones. As far as Unix file-system security is concerned, there are many such features that are well-known to experts, but may surprise uninitiated users.

Subsequently, we consider an example that is not so exotic after all. As may be easily experienced on a running Unix system, the following sequence of commands may put a user's file-system into an uncouth state. Below we assume that `user1` and `user2` are working within the same directory (e.g. somewhere within the home of `user1`).

```
user1> umask 000; mkdir foo
user2> umask 022; mkdir foo/bar
user2> touch foo/bar/baz
```

That is, `user1` creates a directory that is writable for everyone, and `user2` puts there a non-empty directory without write access for others.

In this situation it has become impossible for `user1` to remove his very own directory `foo` without cooperation of `user2`, since `foo` contains another non-empty and non-writable directory, which cannot be removed just now.

```
user1> rmdir foo
rmdir: directory "foo": Directory not empty
user1> rmdir foo/bar
rmdir: directory "bar": Directory not empty
user1> rm foo/bar/baz
rm not removed: Permission denied
```

Only after `user2` has cleaned up his directory `bar`, is `user1` enabled to remove both `foo/bar` and `foo`. Alternatively, `user2` could remove `foo/bar` as well. In the unfortunate case that `user2` does not cooperate or happens to be temporarily unavailable, `user1` would have to find the super user (`root`) to clean up the

situation. In Unix `root` may perform any file-system operation without any access control limitations.

This is the typical Unix way of handling abnormal situations: while it is easy to run into odd cases due to simplistic policies it is as well quite easy to get out. There are other well-known systems around that make it somewhat harder to get into trouble, but almost impossible to escape again!

Is there really no escape for `user1`? Experiments can only show possible ways, but never demonstrate the absence of other means exhaustively. This is a typical situation where (formal) proof may help. Subsequently, we model the main aspects Unix file-system security within the Isabelle/HOL environment (cf. chapter 7) and prove that there is indeed no way for `user1` to get rid of his directory `foo` without help by others (see §10.6.4 for the main theorem stating this).

10.3 Unix file-systems

theory *Unix* = *Nested-Environment* + *List-Prefix*:[1]

We give a simple mathematical model of the basic structures underlying the Unix file-system, together with a few fundamental operations that could be imagined to be performed internally by the Unix kernel. This forms the basis for the set of Unix system-calls to be introduced later on (see §10.4), which are the actual interface offered to processes running in user-space.

Basically, any Unix file is either a *plain file* or a *directory*, consisting of some *content* plus *attributes*. The content of a plain file is plain text. The content of a directory is a mapping from names to further files. In fact, this is the only way that names get associated with files. In Unix files do *not* have a name in itself. Even more, any number of names may be associated with the very same file due to *hard links* (although this is not handled in our model). Attributes include information to control various ways to access the file (read, write etc.).

Our model will be quite liberal in omitting excessive detail that is easily seen to be "irrelevant" for the aspects of Unix file-systems to be discussed here. First of all, we ignore character and block special files, pipes, sockets, hard links, symbolic links, and mount points.

10.3.1 Names

User ids and file name components shall be represented by natural numbers (without loss of generality). We do not bother about encoding of actual names (e.g. strings), nor a mapping between user names and user ids as would be present in reality.

[1] Theories *Nested-Environment* and *List-Prefix* are included from [Bauer *et al.*, 2001].

types
uid = *nat*
name = *nat*
path = *name list*

10.3.2 Attributes

Unix file attributes mainly consist of *owner* information and *permission* bits, which control access for "user", "group", and "others" (see also the Unix man pages *chmod*(2) and *stat*(2) for details).

Our model of file permissions only considers the "others" part. The "user" field may be omitted without loss of generality from the security point of view, since the owner is usually able to change it anyway by performing *chmod*. We omit "group" permissions as a genuine simplification, since we just do not intend to discuss a model of multiple groups and group membership here, but pretend that everyone is member of a single global group. A general HOL model of user and group structures is given in [Naraschewski, 2001].

datatype *perm* =
  Readable
| *Writable*
| *Executable* — ignored

types *perms* = *perm set*

record *att* =
  owner :: *uid*
  others :: *perms*

For plain files *Readable* and *Writable* specify read and write access to the actual content, i.e. the string of text stored here. For directories *Readable* determines if the set of entry names may be accessed, and *Writable* controls the ability to create or delete any entries (both plain files or sub-directories).

As another simplification, we ignore the *Executable* permission altogether. In reality it would indicate executable plain files (also known as "binaries"), or control actual retrieval of directory entries (recall that mere directory browsing is controlled via *Readable*). Note that the latter means that in order to perform any file-system operation whatsoever, all directories encountered on the path would have to grant *Executable*. We ignore this detail and pretend that all directories give *Executable* permission to anybody, which is usually the case in real-world file-systems anyway.

10.3.3 Files

In order to model the general tree structure of a Unix file-system we use the arbitrarily branching datatype ($'a$, $'b$, $'c$) *env* from the supplemental library

of Isabelle/HOL [Bauer *et al.*, 2001] (theory *Nested-Environment*). This type provides constructors *Val* and *Env* as follows:

```
Val :: 'a ⇒ ('a, 'b, 'c) env
Env :: 'b ⇒ ('c ⇒ ('a, 'b, 'c) env option) ⇒ ('a, 'b, 'c) env
```

Here the parameter $'a$ refers to basic information occurring at leaf positions, parameter $'b$ to information kept with inner branch nodes, and parameter $'c$ to the index type of the tree structure. For our purpose we use the type instance with $att \times string$ (representing plain files), *att* (for attributes of directory nodes), and *name* (for the index type of directory nodes).

types
```
file = (att × string, att, name) env
```

The theory also provides *lookup* and *update* operations for general tree structures with the subsequent primitive recursive characterizations.

```
lookup :: ('a, 'b, 'c) env ⇒ 'c list ⇒ ('a, 'b, 'c) env option
update :: 'c list ⇒ ('a, 'b, 'c) env option ⇒ ('a, 'b, 'c) env ⇒ ('a, 'b, 'c) env

lookup env xs =
(case xs of [] ⇒ Some env
 | x # xs ⇒
    case env of Val a ⇒ None
    | Env b es ⇒ case es x of None ⇒ None | Some e ⇒ lookup e xs)

update xs opt env =
(case xs of [] ⇒ case opt of None ⇒ env | Some e ⇒ e
 | x # xs ⇒
    case env of Val a ⇒ Val a
    | Env b es ⇒
       case xs of [] ⇒ Env b (es(x := opt))
       | y # ys ⇒
          Env b
           (es(x := case es x of None ⇒ None
                    | Some e ⇒ Some (update (y # ys) opt e))))
```

Several basic properties of these operations are proven in the same theory. These will be routinely used later on without further notice.

Apparently, the elements of type *file* contain an *att* component in either case. Subsequently, we define a few auxiliary operations to manipulate this field uniformly, following the conventions for record types in Isabelle/HOL (cf. §7.2.3).

constdefs
```
attributes :: file ⇒ att
attributes file ≡
  (case file of
```

```
    Val (att, text) ⇒ att
  | Env att dir ⇒ att)

 attributes-update :: att ⇒ file ⇒ file
 attributes-update att file ≡
  (case file of
    Val (att', text) ⇒ Val (att, text)
  | Env att' dir ⇒ Env att dir)
```

lemma [*simp*]: *attributes* (*Val* (*att*, *text*)) = *att*
by (*simp add*: *attributes-def*)

lemma [*simp*]: *attributes* (*Env att dir*) = *att*
by (*simp add*: *attributes-def*)

lemma [*simp*]: (*Val* (*att*, *text*)) (|*attributes* := *att'*|) = *Val* (*att'*, *text*)
by (*simp add*: *attributes-update-def*)

lemma [*simp*]: (*Env att dir*) (|*attributes* := *att'*|) = *Env att' dir*
by (*simp add*: *attributes-update-def*)

lemma [*simp*]: *attributes* (*file* (|*attributes* := *att*|)) = *att*
by (*cases file*) (*simp-all add*: *attributes-def split-tupled-all*)

10.3.4 Initial file-systems

Given a set of *known users* a file-system shall be initialized by providing an empty home directory for each user, with read-only access for everybody else. Note that we may directly use the user id as home directory name, since both types have been identified. Certainly, the very root directory is owned by the super user (who has user id 0).

constdefs

```
 init :: uid set ⇒ file
 init users ≡
  Env (|owner = 0, others = {Readable}|)
   (λu. if u ∈ users then Some (Env (|owner = u, others = {Readable}|) empty)
    else None)
```

10.3.5 Accessing file-systems

The main internal file-system operation is access of a file by a user, requesting a certain set of permissions. The resulting *file option* indicates if the file had been present at the corresponding *path* and if access was granted according to the permissions recorded within the file-system.

Note that by the rules of Unix file-system security (cf. [Tanenbaum, 1992] and our simplifications outlined before), both the super-user and owner may always access a file unconditionally.

```
constdefs
  access :: file ⇒ path ⇒ uid ⇒ perms ⇒ file option
  access root path uid perms ≡
    (case lookup root path of
      None ⇒ None
    | Some file ⇒
        if uid = 0
          ∨ uid = owner (attributes file)
          ∨ perms ⊆ others (attributes file)
        then Some file
        else None)
```

Successful access to a certain file is the main prerequisite for system-calls to be applicable (see also §10.4). Any modification of the file-system is then performed using the basic *update* operation of the nested environment type.

Apparently, *access* is just a wrapper for the basic *lookup* function, with additional checking of attributes. Subsequently we establish a few auxiliary facts that stem from the primitive *lookup* used within *access*. The notion of independent paths "∥" is defined in theory *List-Prefix* of [Bauer *et al.*, 2001].

```
lemma access-empty-lookup: access root path uid {} = lookup root path
  by (simp add: access-def split: option.splits)

lemma access-some-lookup:
  access root path uid perms = Some file ⟹
    lookup root path = Some file
  by (simp add: access-def split: option.splits if-splits)

lemma access-update-other: path' ∥ path ⟹
  access (update path' opt root) path uid perms = access root path uid perms
proof −
  assume path' ∥ path
  then obtain y z xs ys zs where
      y ≠ z and path' = xs @ y # ys and path = xs @ z # zs
    by (blast dest: parallel-decomp)
  hence lookup (update path' opt root) path = lookup root path
    by (blast intro: lookup-update-other)
  thus ?thesis by (simp only: access-def)
qed
```

10.4 File-system transitions

10.4.1 Unix system calls

According to established operating system design (cf. [Tanenbaum, 1992]) user space processes may only initiate system operations by a fixed set of *system-calls*. This principle enables the kernel to enforce certain security policies in

the first place. This is essentially the very same idea employed by "LCF-style" theorem proving systems according to Milner's principle of "Correctness by Construction", such as Isabelle itself.

In our model of Unix we give a fixed datatype *operation* for the syntax of system-calls, together with a relation of file-system state transitions $root\ -x\rightarrow\ root'$ as operational semantics introduced later on.

datatype *operation* =
  Read uid string path
| *Write uid string path*
| *Chmod uid perms path*
| *Creat uid perms path*
| *Unlink uid path*
| *Mkdir uid perms path*
| *Rmdir uid path*
| *Readdir uid name set path*

The *uid* field of an operation corresponds to the *effective user id* of the underlying process, although our model never mentions processes explicitly. The other parameters are provided as arguments by the caller; the *path* is common to all kinds of system-calls.

consts
  uid-of :: *operation* ⇒ *uid*
primrec
  uid-of (*Read uid text path*) = *uid*
  uid-of (*Write uid text path*) = *uid*
  uid-of (*Chmod uid perms path*) = *uid*
  uid-of (*Creat uid perms path*) = *uid*
  uid-of (*Unlink uid path*) = *uid*
  uid-of (*Mkdir uid path perms*) = *uid*
  uid-of (*Rmdir uid path*) = *uid*
  uid-of (*Readdir uid names path*) = *uid*

consts
  path-of :: *operation* ⇒ *path*
primrec
  path-of (*Read uid text path*) = *path*
  path-of (*Write uid text path*) = *path*
  path-of (*Chmod uid perms path*) = *path*
  path-of (*Creat uid perms path*) = *path*
  path-of (*Unlink uid path*) = *path*
  path-of (*Mkdir uid perms path*) = *path*
  path-of (*Rmdir uid path*) = *path*
  path-of (*Readdir uid names path*) = *path*

Note that we have omitted explicit *Open* and *Close* operations, pretending that

Read and *Write* would already take care of this behind the scenes. Thus we have basically treated sequences of real system-calls *open–read/write–close* as atomic transactions.

In principle, this simplification could make big a difference in a model with explicit concurrent processes. On the other hand, on a real Unix system the exact scheduling of concurrent *open* and *close* operations does *not* directly affect the success of corresponding *read* or *write*. Unix allows several processes to have files opened at the same time, even for writing [Ritchie and Thompson, 1974]. Certainly, the result from reading the contents later on may be hard to predict, but the system-calls involved here will succeed unconditionally.

The operational semantics of system calls is now specified via transitions of the file-system configuration. This is expressed as an inductive relation, although there is no recursion involved here.

consts
transition :: (*file* × *operation* × *file*) *set*

syntax
-*transition* :: *file* ⇒ *operation* ⇒ *file* ⇒ *bool*
(- −-⟶ - [90, 1000, 90] 100)
translations
root −*x*⟶ *root′* ⇌ (*root*, *x*, *root′*) ∈ *transition*

inductive *transition*
intros

read:
access root path uid {*Readable*} = *Some* (*Val* (*att*, *text*)) ⟹
root −(*Read uid text path*)⟶ *root*
write:
access root path uid {*Writable*} = *Some* (*Val* (*att*, *text′*)) ⟹
root −(*Write uid text path*)⟶ *update path* (*Some* (*Val* (*att*, *text*))) *root*

chmod:
access root path uid {} = *Some file* ⟹
uid = 0 ∨ *uid* = *owner* (*attributes file*) ⟹
root −(*Chmod uid perms path*)⟶ *update path*
(*Some* (*file* (|*attributes* := *attributes file* (|*others* := *perms*|)|))) *root*

creat:
path = *parent-path* @ [*name*] ⟹
access root parent-path uid {*Writable*} = *Some* (*Env att parent*) ⟹
access root path uid {} = *None* ⟹
root −(*Creat uid perms path*)⟶ *update path*
(*Some* (*Val* ((|*owner* = *uid*, *others* = *perms*|), []))) *root*

unlink:
  path = *parent-path* @ [*name*] ⟹
    access root parent-path uid {*Writable*} = *Some* (*Env att parent*) ⟹
    access root path uid {} = *Some* (*Val plain*) ⟹
    root −(*Unlink uid path*)→ *update path None root*

mkdir:
  path = *parent-path* @ [*name*] ⟹
    access root parent-path uid {*Writable*} = *Some* (*Env att parent*) ⟹
    access root path uid {} = *None* ⟹
    root −(*Mkdir uid perms path*)→ *update path*
      (*Some* (*Env* (|*owner* = *uid*, *others* = *perms*|) *empty*)) *root*
rmdir:
  path = *parent-path* @ [*name*] ⟹
    access root parent-path uid {*Writable*} = *Some* (*Env att parent*) ⟹
    access root path uid {} = *Some* (*Env att′ empty*) ⟹
    root −(*Rmdir uid path*)→ *update path None root*

readdir:
  access root path uid {*Readable*} = *Some* (*Env att dir*) ⟹
    names = *dom dir* ⟹
    root −(*Readdir uid names path*)→ *root*

The above specification is central to the whole formal development. Any of the results to be established later on are only meaningful to the outside world if this transition system provides an adequate model of real Unix systems. This kind of "reality-check" of a formal model is the well-known problem of *validation*.

In case of doubt, one may consider to compare our definition with the informal specifications given in the corresponding Unix man pages, or inspect the sources of a real implementation such as [Torvalds and others]. Another common way to gain confidence into our formal model is to run simple simulations (see §10.5.2), and compare the results with those of experiments performed on a running Unix system, for example.

10.4.2 Basic properties of single transitions

The transition system *root* −*x*→ *root′* defined above determines a unique result *root′* from given *root* and *x* (this holds rather trivially, as there is only one clause for each operation). This uniqueness statement will simplify our subsequent development to some extent, since we only have to reason about a partial function rather than a general relation.

theorem *transition-uniq*: *root* −*x*→ *root′* ⟹ *root* −*x*→ *root″* ⟹ *root′* = *root″*
proof −
  assume *root*: *root* −*x*→ *root′*
  assume *root* −*x*→ *root″*

```
  thus ?thesis
  proof cases
    case read
    with root show ?thesis by cases auto
  next
    case write
    with root show ?thesis by cases auto
  next
    case chmod
    with root show ?thesis by cases auto
  next
    case creat
    with root show ?thesis by cases auto
  next
    case unlink
    with root show ?thesis by cases auto
  next
    case mkdir
    with root show ?thesis by cases auto
  next
    case rmdir
    with root show ?thesis by cases auto
  next
    case readdir
    with root show ?thesis by cases auto
  qed
qed
```

Apparently, file-system transitions are *type-safe* in the sense that the result of transforming an actual directory yields again a directory.

```
theorem transition-type-safe:
  root −x→ root′ ⟹ ∃ att dir. root = Env att dir ⟹ ∃ att dir. root′ = Env att dir
proof −
  assume tr: root −x→ root′
  assume inv: ∃ att dir. root = Env att dir
  show ?thesis
  proof (cases path-of x)
    case Nil
    with tr inv show ?thesis
      by cases (auto simp add: access-def split: if-splits)
  next
    case Cons
    from tr obtain opt where
        root′ = root ∨ root′ = update (path-of x) opt root
      by cases auto
```

```
    with inv Cons show ?thesis
      by (auto simp add: update-eq split: list.splits)
  qed
qed
```

The previous result may be seen as the most basic invariant on the file-system state that is enforced by any proper Unix kernel implementation. So user processes — being bound to the system-call interface — may never mess up a file-system such that the root becomes a plain file instead of a directory, which would be a very odd configuration of the file-system indeed.

10.4.3 Iterated transitions

Iterated file-system transitions via finite sequences of system operations are modeled inductively as follows.

```
consts
  transitions :: (file × operation list × file) set

syntax
  -transitions :: file ⇒ operation list ⇒ file ⇒ bool
  (- =-⇒ - [90, 1000, 90] 100)
translations
  root =xs⇒ root' ⇌ (root, xs, root') ∈ transitions

inductive transitions
  intros
    nil: root =[]⇒ root
    cons: root −x→ root' ⟹ root' =xs⇒ root'' ⟹ root =(x # xs)⇒ root''
```

In a sense, this relation describes the cumulative effect of the sequence of system-calls issued by a set of running processes over a finite number of run-time steps.

We establish a few basic facts relating iterated transitions with single ones, according to the recursive structure of lists.

```
lemma transitions-nil-eq: root =[]⇒ root' = (root = root')
proof
  assume root =[]⇒ root'
  thus root = root' by cases simp-all
next
  assume root = root'
  thus root =[]⇒ root' by (simp only: transitions.nil)
qed

lemma transitions-cons-eq:
  root =(x # xs)⇒ root'' = (∃ root'. root −x→ root' ∧ root' =xs⇒ root'')
proof
```

```
  assume root =(x # xs)⇒ root''
  thus ∃root'. root −x→ root' ∧ root' =xs⇒ root''
    by cases auto
next
  assume ∃root'. root −x→ root' ∧ root' =xs⇒ root''
  thus root =(x # xs)⇒ root''
    by (blast intro: transitions.cons)
qed
```

The next two rules show how to "destruct" known transition sequences. Note that the second one actually relies on the uniqueness property of the basic transition system (cf. §10.4.2).

```
lemma transitions-nilD: root =[]⇒ root' ⟹ root' = root
  by (simp add: transitions-nil-eq)

lemma transitions-consD:
  root =(x # xs)⇒ root'' ⟹ root −x→ root' ⟹ root' =xs⇒ root''
proof −
  assume root =(x # xs)⇒ root''
  then obtain r' where r': root −x→ r' and root'': r' =xs⇒ root''
    by cases simp-all
  assume root −x→ root'
  with r' have r' = root' by (rule transition-uniq)
  with root'' show root' =xs⇒ root'' by simp
qed
```

The following fact shows how an invariant Q of single transitions with satisfying P may be transferred to iterated transitions. The proof is rather obvious by rule induction according to the definition of *root* $=xs\Rightarrow$ *root'*.

```
lemma transitions-invariant:
  (⋀r x r'. r −x→ r' ⟹ Q r ⟹ P x ⟹ Q r') ⟹
    root =xs⇒ root' ⟹ Q root ⟹ ∀x ∈ set xs. P x ⟹ Q root'
proof −
  assume r: ⋀r x r'. r −x→ r' ⟹ Q r ⟹ P x ⟹ Q r'
  assume root =xs⇒ root'
  thus Q root ⟹ (∀x ∈ set xs. P x) ⟹ Q root' (is PROP ?P root xs root')
  proof (induct ?P root xs root')
    fix root assume Q root
    thus Q root .
  next
    fix root root' root'' and x xs
    assume root': root −x→ root'
    assume hyp: PROP ?P root' xs root''
    assume Q: Q root
    assume P: ∀x ∈ set (x # xs). P x
```

```
    hence P x by simp
    with root' Q have Q root' by (rule r)
    moreover from P have ∀x ∈ set xs. P x by simp
    ultimately show Q root'' by (rule hyp)
  qed
qed
```

As a basic application of the previous result, we transfer the type-safety property (§10.4.2) from single transitions to iterated ones.

```
theorem transitions-type-safe:
  root =xs⇒ root' ⟹ ∃att dir. root = Env att dir ⟹ ∃att dir. root' = Env att dir
proof −
  case antecedent
  with transition-type-safe show ?thesis
  proof (rule transitions-invariant)
    show ∀x ∈ set xs. True by blast
  qed
qed
```

10.5 Executable sequences

An inductively defined relation such as $root\ -x\rightarrow\ root'$ (cf. §10.4.1) has two main aspects. First of all, the resulting system admits a certain set of transition rules (introductions) as given in the specification. Secondly, there is a least fixed-point construction involved, which results in induction (and case-analysis) rules to eliminate known transitions exhaustively.

Subsequently, we explore our transition system in an experimental manner, mainly using the introduction rules with basic algebraic properties of the underlying structures. This technique closely resembles that of logic programming combined with functional evaluation in a very simple manner.

Just as the "closed-world assumption" is left implicit in logic programming, we do not refer to induction over the whole transition system here. So this is still positive reasoning only, about possible executions; exhaustive reasoning will be employed only later on (see §10.6), when we shall demonstrate that certain behavior is *not* possible.

10.5.1 Possible transitions

Rather obviously, a list of system operations can be executed within a certain state if there is a result state reached by an iterated transition.

```
constdefs
  can-exec :: file ⇒ operation list ⇒ bool
  can-exec root xs ≡ ∃root'. root =xs⇒ root'
```

lemma *can-exec-nil*: *can-exec root* []
by (*unfold can-exec-def*) (*blast intro*: *transitions.intros*)

lemma *can-exec-cons*:
root $-x\rightarrow$ *root'* $\Longrightarrow$ *can-exec root' xs* $\Longrightarrow$ *can-exec root* (*x* # *xs*)
by (*unfold can-exec-def*) (*blast intro*: *transitions.intros*)

In case that we already know that a sequence can be executed we may destruct it backwards into individual transitions.

lemma *can-exec-snocD*: $\bigwedge$*root*. *can-exec root* (*xs* @ [*y*])
$\Longrightarrow$ $\exists$ *root' root''*. *root* $=$*xs*$\Rightarrow$ *root'* $\wedge$ *root'* $-y\rightarrow$ *root''*
(**is** *PROP ?P xs* **is** $\bigwedge$*root*. *?A root xs* $\Longrightarrow$ *?C root xs*)
proof (*induct xs*)
fix *root*
{
assume *?A root* []
thus *?C root* []
by (*simp add*: *can-exec-def transitions-nil-eq transitions-cons-eq*)
next
fix *x xs*
assume *hyp*: *PROP ?P xs*
assume *asm*: *?A root* (*x* # *xs*)
show *?C root* (*x* # *xs*)
proof −
from *asm* **obtain** *r root''* **where** *x*: *root* $-x\rightarrow$ *r* **and**
xs-y: *r* $=$(*xs* @ [*y*])$\Rightarrow$ *root''*
by (*auto simp add*: *can-exec-def transitions-nil-eq transitions-cons-eq*)
from *xs-y hyp* **obtain** *root' r'* **where** *xs*: *r* $=$*xs*$\Rightarrow$ *root'* **and** *y*: *root'* $-y\rightarrow$ *r'*
by (*auto simp add*: *can-exec-def*)
from *x xs* **have** *root* $=$(*x* # *xs*)$\Rightarrow$ *root'*
by (*rule transitions.cons*)
with *y* **show** *?thesis* **by** *blast*
qed
}
qed

10.5.2 Example executions

We are ready to perform a few experiments within our formal model of Unix system-calls. The common technique is to alternate introduction rules of the transition system (§10.4), and steps to solve any emerging side conditions by using algebraic properties of the underlying file-system structures (§10.3). Note that this does not constitute "real proof", but essentially performs symbolic evaluation within the logical environment.

lemmas *eval* = *access-def init-def*

theorem $u \in users \Longrightarrow can\text{-}exec\ (init\ users)$
$[Mkdir\ u\ perms\ [u,\ name]]$
apply (*rule can-exec-cons*)
— back-chain *can-exec* (of *Cons*)
apply (*rule mkdir*)
— back-chain *Mkdir*
apply (*force simp add*: *eval*)+
— solve preconditions of *Mkdir*
apply (*simp add*: *eval*)
— peek at normalized result (optional)

1. $u \in users \Longrightarrow$
$can\text{-}exec$
$(Env\ (\!|owner = 0,\ others = \{Readable\}|\!)$
$((\lambda u.\ if\ u \in users$
$then\ Some\ (Env\ (\!|owner = u,\ others = \{Readable\}|\!)\ empty)$
$else\ None)$
$(u \mapsto Env\ (\!|owner = u,\ others = \{Readable\}|\!)$
$(empty(name \mapsto Env\ (\!|owner = u,\ others = perms|\!)\ empty)))))$
$[]$

apply (*rule can-exec-nil*)
— back-chain *can-exec* (of *Nil*)
done

By inspecting the result shown just before the final step above, we may gain confidence that our specification of Unix system-calls actually makes sense. Further common errors are usually exhibited when preconditions of transition rules fail unexpectedly.

Here are some additional experiments, using the same techniques as before.

theorem $u \in users \Longrightarrow can\text{-}exec\ (init\ users)$
$[Creat\ u\ perms\ [u,\ name],$
$Unlink\ u\ [u,\ name]]$
apply (*rule can-exec-cons*)
apply (*rule creat*)
apply (*force simp add*: *eval*)+
apply (*simp add*: *eval*)
apply (*rule can-exec-cons*)
apply (*rule unlink*)
apply (*force simp add*: *eval*)+
apply (*simp add*: *eval*)

peek at result:

1. $u \in users \Longrightarrow$
$can\text{-}exec$

```
      (Env (|owner = 0, others = {Readable}|)
        ((λu. if u ∈ users
              then Some (Env (|owner = u, others = {Readable}|) empty)
              else None)
          (u↦Env (|owner = u, others = {Readable}|) empty)))
      []

  apply (rule can-exec-nil)
  done

theorem u ∈ users ⟹ Writable ∈ perms1 ⟹
  Readable ∈ perms2 ⟹ name3 ≠ name4 ⟹
  can-exec (init users)
   [Mkdir u perms1 [u, name1],
    Mkdir u' perms2 [u, name1, name2],
    Creat u' perms3 [u, name1, name2, name3],
    Creat u' perms3 [u, name1, name2, name4],
    Readdir u {name3, name4} [u, name1, name2]]
  apply (rule can-exec-cons, rule transition.intros,
    (force simp add: eval)+, (simp add: eval)?)+
```

peek at result:

```
  1. u ∈ users ⟹
     Writable ∈ perms1 ⟹
     Readable ∈ perms2 ⟹
     name3 ≠ name4 ⟹
     can-exec
      (Env (|owner = 0, others = {Readable}|)
        ((λu. if u ∈ users
              then Some (Env (|owner = u, others = {Readable}|) empty)
              else None)
          (u↦Env (|owner = u, others = {Readable}|)
              (empty(name1↦
               Env (|owner = u, others = perms1|)
                (empty(name2↦
                 Env (|owner = u', others = perms2|)
                  (empty(name3↦Val ((|owner = u', others = perms3|), []))
                   (name4↦Val ((|owner = u', others = perms3|), [])))))))))
      []

  apply (rule can-exec-nil)
  done

theorem u ∈ users ⟹ Writable ∈ perms1 ⟹ Readable ∈ perms3 ⟹
  can-exec (init users)
   [Mkdir u perms1 [u, name1],
    Mkdir u' perms2 [u, name1, name2],
```

```
  Creat u' perms3 [u, name1, name2, name3],
  Write u' ''foo'' [u, name1, name2, name3],
  Read u ''foo'' [u, name1, name2, name3]]
 apply (rule can-exec-cons, rule transition.intros,
  (force simp add: eval)+, (simp add: eval)?)+
```

peek at result:

```
 1. u ∈ users ⟹
    Writable ∈ perms1 ⟹
    Readable ∈ perms3 ⟹
    can-exec
     (Env (|owner = 0, others = {Readable}|)
       ((λu. if u ∈ users
             then Some (Env (|owner = u, others = {Readable}|) empty)
             else None)
        (u↦Env (|owner = u, others = {Readable}|)
             (empty(name1↦
              Env (|owner = u, others = perms1|)
               (empty(name2↦
                Env (|owner = u', others = perms2|)
                 (empty(name3↦
                  Val ((|owner = u', others = perms3|), ''foo''))))))))))
    []

 apply (rule can-exec-nil)
 done
```

10.6 Odd effects — treated formally

We are ready to give a completely formal treatment of the odd situation discussed in the introduction (§10.2): the file-system may reach a state where a user is unable to remove his very own directory, because it is still populated by items placed there by another user in an uncouth manner.

10.6.1 The general procedure

The following theorem expresses the general procedure we are following to achieve the main result.

```
theorem general-procedure:
 (⋀r r'. Q r ⟹ r −y→ r' ⟹ False) ⟹
   (⋀root. init users =bs⇒ root ⟹ Q root) ⟹
   (⋀r x r'. r −x→ r' ⟹ Q r ⟹  P x ⟹ Q r') ⟹
   init users =bs⇒ root ⟹ ¬ (∃xs. (∀x ∈ set xs. P x) ∧ can-exec root (xs @ [y]))
proof −
 assume cannot-y: ⋀r r'. Q r ⟹ r −y→ r' ⟹ False
```

```
  assume init-inv: ⋀root. init users =bs⇒ root ⟹ Q root
  assume preserve-inv: ⋀r x r'. r −x→ r' ⟹ Q r ⟹ P x ⟹ Q r'
  assume init-result: init users =bs⇒ root
  {
    fix xs
    assume Ps: ∀ x ∈ set xs. P x
    assume can-exec: can-exec root (xs @ [y])
    then obtain root' root'' where
        xs: root =xs⇒ root' and y: root' −y→ root''
      by (blast dest: can-exec-snocD)
    from init-result have Q root by (rule init-inv)
    from preserve-inv xs this Ps have Q root'
      by (rule transitions-invariant)
    from this y have False by (rule cannot-y)
  }
  thus ?thesis by blast
qed
```

Here $P\ x$ refers to the restriction on file-system operations that are admitted after having reached the critical configuration; according to the problem specification this will become $\mathit{uid\text{-}of}\ x = \mathit{user}_1$ later on. Furthermore, y refers to the operations we claim to be impossible to perform afterwards; we will be taking *Rmdir*. Moreover, Q is a suitable (auxiliary) invariant over the file-system (after reaching the critical configuration); choosing Q adequately is very important to make the proof work (see §10.6.3).

10.6.2 The particular setup

We introduce a few global declarations and axioms to describe our particular setup considered here. Thus we avoid excessive use of local parameters in the subsequent development.

```
consts
  users :: uid set
  user₁ :: uid
  user₂ :: uid
  name₁ :: name
  name₂ :: name
  name₃ :: name
  perms₁ :: perms
  perms₂ :: perms

axioms
  user₁-known: user₁ ∈ users
  user₁-not-root: user₁ ≠ 0
  users-neq: user₁ ≠ user₂
```

perms$_1$*-writable*: *Writable* $\in$ *perms*$_1$
perms$_2$*-not-writable*: *Writable* $\notin$ *perms*$_2$

lemmas *setup* =
user$_1$*-known user*$_1$*-not-root users-neq*
perms$_1$*-writable perms*$_2$*-not-writable*

The *bogus* operations are the ones that lead into the uncouth situation described before; *bogus-path* is the key position within the file-system where things go awry.

constdefs
bogus :: *operation list*
bogus $\equiv$
[*Mkdir user*$_1$ *perms*$_1$ [*user*$_1$, *name*$_1$],
Mkdir user$_2$ *perms*$_2$ [*user*$_1$, *name*$_1$, *name*$_2$],
Creat user$_2$ *perms*$_2$ [*user*$_1$, *name*$_1$, *name*$_2$, *name*$_3$]]

bogus-path :: *path*
bogus-path $\equiv$ [*user*$_1$, *name*$_1$, *name*$_2$]

10.6.3 Invariance lemmas

The following invariant over the root file-system describes the bogus situation in an abstract manner: located at a certain *path* (within the home directory of *user*$_1$) is a non-empty directory that is neither owned and nor writable by *user*$_1$.

constdefs
invariant :: *file* $\Rightarrow$ *path* $\Rightarrow$ *bool*
invariant root path $\equiv$
($\exists$ *att dir*.
access root path user$_1$ {} = *Some* (*Env att dir*) $\wedge$ *dir* $\neq$ *empty* $\wedge$
user$_1$ $\neq$ *owner att* $\wedge$
access root path user$_1$ {*Writable*} = *None*)

Following the general procedure (cf. §10.6.1), we will now establish the three key lemmas required to yield the final result.

1. The invariant is sufficiently strong to entail the pathological case that *user*$_1$ is unable to remove the (owned) directory at [*user*$_1$, *name*$_1$].
2. The invariant does hold after having executed the *bogus* list of operations (starting with an initial file-system configuration).
3. The invariant is preserved by any file-system operation performed by *user*$_1$ alone, without any help by other users.

Since the invariant appears both as assumption and conclusion in the course of reasoning, its formulation is rather critical for the whole development to work

out properly. In particular, the third step is very sensitive to the invariant being either too strong or too weak. Moreover, the statement has to be sufficiently abstract, lest the proof become cluttered by confusing detail.

The first two lemmas are straight-forward, we just have to inspect rather special cases.

```
lemma cannot-rmdir: invariant root bogus-path ⟹
  root −(Rmdir user₁ [user₁, name₁])→ root′ ⟹ False
proof −
  assume invariant root bogus-path
  then obtain file where access root bogus-path user₁ {} = Some file
    by (unfold invariant-def) blast
  moreover
  assume root −(Rmdir user₁ [user₁, name₁])→ root′
  then obtain att where
      access root [user₁, name₁] user₁ {} = Some (Env att empty)
    by cases auto
  hence access root ([user₁, name₁] @ [name₂]) user₁ {} = None
    by (simp only: access-empty-lookup lookup-append-some) simp
  ultimately show False by (simp add: bogus-path-def)
qed
```

Subsequently, we use again the same techniques for symbolic evaluation as encountered in §10.5.2.

```
lemma init-invariant: init users =bogus⇒ root ⟹ invariant root bogus-path
proof −
  note eval = setup access-def init-def
  case antecedent thus ?thesis
    apply (unfold bogus-def bogus-path-def)
    apply (drule transitions-consD, rule transition.intros,
      (force simp add: eval)+, (simp add: eval)?)+
      — evaluate all operations
    apply (drule transitions-nilD)
      — reach final result
    apply (simp add: invariant-def eval)
      — check the invariant
    done
qed
```

Finally we are left with the main effort to show that the "bogosity" invariant is preserved by any file-system operation $root\ -x\rightarrow\ root'$ performed by $user_1$ alone. Note that this holds for any *path* given, the particular *bogus-path* is not required here.

```
lemma preserve-invariant: root −x→ root′ ⟹
  invariant root path ⟹ uid-of x = user₁ ⟹ invariant root′ path
```

```
proof −
  assume tr: root −x→ root′
  assume inv: invariant root path
  assume uid: uid-of x = user₁

  from inv obtain att dir where
      inv₁: access root path user₁ {} = Some (Env att dir) and
      inv₂: dir ≠ empty and
      inv₃: user₁ ≠ owner att and
      inv₄: access root path user₁ {Writable} = None
    by (auto simp add: invariant-def)

  from inv₁ have lookup: lookup root path = Some (Env att dir)
    by (simp only: access-empty-lookup)

  from inv₁ inv₃ inv₄ and user₁-not-root
  have not-writable: Writable ∉ others att
    by (auto simp add: access-def split: option.splits if-splits)

  show ?thesis
  proof cases
    assume root′ = root
    with inv show invariant root′ path by (simp only:)
  next
    assume changed: root′ ≠ root
    with tr obtain opt where root′: root′ = update (path-of x) opt root
      by cases auto
    show ?thesis
    proof (rule prefix-cases)
      assume path-of x ∥ path
      with inv root′
      have ⋀perms. access root′ path user₁ perms = access root path user₁ perms
        by (simp only: access-update-other)
      with inv show invariant root′ path
        by (simp only: invariant-def)
    next
      assume path-of x ≤ path
      then obtain ys where path: path = path-of x @ ys ..

      show ?thesis
      proof (cases ys)
        assume ys = []
        with tr uid inv₂ inv₃ lookup changed path and user₁-not-root
        have False
          by cases (auto simp add: access-empty-lookup dest: access-some-lookup)
        thus ?thesis ..
      next
        fix z zs assume ys: ys = z # zs
        have lookup root′ path = lookup root path
        proof −
```

```
      from inv₂ lookup path ys
      have look: lookup root (path-of x @ z # zs) = Some (Env att dir)
        by (simp only:)
      then obtain att' dir' file' where
          look': lookup root (path-of x) = Some (Env att' dir') and
          dir': dir' z = Some file' and
          file': lookup file' zs = Some (Env att dir)
        by (blast dest: lookup-some-upper)

      from tr uid changed look' dir' obtain att'' where
          look'': lookup root' (path-of x) = Some (Env att'' dir')
        by cases (auto simp add: access-empty-lookup lookup-update-some
          dest: access-some-lookup)
      with dir' file' have lookup root' (path-of x @ z # zs) =
          Some (Env att dir)
        by (simp add: lookup-append-some)
      with look path ys show ?thesis
        by simp
    qed
    with inv show invariant root' path
      by (simp only: invariant-def access-def)
  qed
next
  assume path < path-of x
  then obtain y ys where path: path-of x = path @ y # ys ..

  obtain dir' where
    lookup': lookup root' path = Some (Env att dir') and
    inv₂': dir' ≠ empty
  proof (cases ys)
    assume ys = []
    with path have parent: path-of x = path @ [y] by simp
    with tr uid inv₄ changed obtain file where
        root' = update (path-of x) (Some file) root
      by cases auto
    with lookup parent have lookup root' path = Some (Env att (dir(y↦file)))
      by (simp only: update-append-some update-cons-nil-env)
    moreover have dir(y↦file) ≠ empty by simp
    ultimately show ?thesis ..
  next
    fix z zs assume ys: ys = z # zs
    with lookup root' path
    have lookup root' path = Some (update (y # ys) opt (Env att dir))
      by (simp only: update-append-some)
    also obtain file' where
      update (y # ys) opt (Env att dir) = Env att (dir(y↦file'))
    proof −
      have dir y ≠ None
      proof −
        have dir y = lookup (Env att dir) [y]
```

```
          by (simp split: option.splits)
        also from lookup have ... = lookup root (path @ [y])
          by (simp only: lookup-append-some)
        also have ... ≠ None
        proof −
          from ys obtain us u where rev-ys: ys = us @ [u]
            by (cases ys rule: rev-cases) auto
          with tr path
          have lookup root ((path @ [y]) @ (us @ [u])) ≠ None ∨
              lookup root ((path @ [y]) @ us) ≠ None
            by cases (auto dest: access-some-lookup)
          thus ?thesis by (blast dest!: lookup-some-append)
        qed
        finally show ?thesis .
      qed
      with ys show ?thesis
        by (insert that, auto simp add: update-cons-cons-env)
    qed
    also have dir(y↦file′) ≠ empty by simp
    ultimately show ?thesis ..
  qed

  from lookup′ have inv₁′: access root′ path user₁ {} = Some (Env att dir′)
    by (simp only: access-empty-lookup)

  from inv₃ lookup′ and not-writable user₁-not-root
  have access root′ path user₁ {Writable} = None
    by (simp add: access-def)
  with inv₁′ inv₂′ inv₃ show ?thesis by (unfold invariant-def) blast
 qed
qed
```

10.6.4 Putting it all together

The main result is now imminent, just by composing the three invariance lemmas (§10.6.3) according the overall procedure (§10.6.1).

```
theorem main:
  init users =bogus⇒ root ⟹
  ¬ (∃ xs. (∀ x ∈ set xs. uid-of x = user₁) ∧
      can-exec root (xs @ [Rmdir user₁ [user₁, name₁]]))
proof −
  case antecedent
  with cannot-rmdir init-invariant preserve-invariant
  show ?thesis by (rule general-procedure)
qed

end
```

10.7 Discussion

10.7.1 Isar techniques

The present Isabelle/Isar application routinely uses advanced techniques discussed in chapter 5 and chapter 6. We reconsider a number of notable instances of advanced proof patterns, as encountered in the present body of text.

Structured treatment of numerous cases

The proof of theorem *transition-uniq* (§10.4.2) proceeds by canonical case-analysis over two independent transitions $root\ -x\rightarrow\ root'$ and $root\ -x\rightarrow\ root''$. Thus we essentially arrive at a quadratic number of sub-problems, stemming from the individual inductive cases of each transition.

This nested case-analysis is arranged in our proof text by first performing an outer backwards decomposition via "**proof** *cases*", and laying out the resulting 8 sub-problems using symbolic case names and term abbreviations (cf. §5.4); each individual sub-problem acquires additional premises stemming from the original **inductive** definition (§10.4.1), which are included in another case analysis: in "**by** *cases auto*" the initial *cases* step splits into 8 new sub-problems (via elimination of the *root* fact), while the terminal *auto* solves all of these uniformly (using the remaining facts of the previous case).

theorem *transition-uniq*: $root\ -x\rightarrow\ root' \Longrightarrow root\ -x\rightarrow\ root'' \Longrightarrow root' = root''$
proof −
  assume *root*: $root\ -x\rightarrow\ root'$
  assume $root\ -x\rightarrow\ root''$
  thus *?thesis*
  proof *cases*
    case *read*
    with *root* **show** *?thesis* **by** *cases auto*
  next
    case *write*
    with *root* **show** *?thesis* **by** *cases auto*
  next
    ⋮

Apparently, we have been able to express this pattern of reasoning quite succinctly as an Isar text, covering the overall structure of the proof, and the main statements and facts with an indication of their use in specific proof steps. Due to the general compositional nature of Isar proofs, this scheme may be easily refined to work out further details of sub-proofs as appropriate (e.g. for special treatment of less obvious cases in more complex applications).

In contrast, we could certainly produce an even shorter script that achieves mostly the same operational behavior, at the cost of the usual disadvantages

of unstructured proof techniques. In particular, modular treatment of sub-problems would be lost; thus detailed analysis of the individual cases and debugging of failed intermediate steps quickly becomes a serious effort.

theorem *transition-uniq*: $root\ -x\rightarrow\ root' \Longrightarrow root\ -x\rightarrow\ root'' \Longrightarrow root' = root''$
by (*erule transition.cases*) (*erule transition.cases*, *auto*)+

Abstract covering of cases

The proof of theorem *transition-type-safe* (§10.4.2) exhibits a different view on case-analysis. Rather than following the superficial structure of cases from the original **inductive** definition of $root\ -x\rightarrow\ root'$ (§10.4.1) naively, we first discriminate against the syntactic structure of the *path-of x* parameter and then inspect the transition.

theorem *transition-type-safe*:
$root\ -x\rightarrow\ root' \Longrightarrow \exists\, att\ dir.\ root = Env\ att\ dir \Longrightarrow \exists\, att\ dir.\ root' = Env\ att\ dir$
proof −
assume *tr*: $root\ -x\rightarrow\ root'$
assume *inv*: $\exists\, att\ dir.\ root = Env\ att\ dir$
show *?thesis*
proof (*cases path-of x*)

At this point we achieve separate cases of *Nil* and *Cons*; both are eventually finished in a similar manner, by performing the inductive case-analysis that has been deferred so far, and solving the remaining problems automatically.

case *Nil*
with *tr inv* **show** *?thesis*
by *cases* (*auto simp add*: *access-def split*: *if-splits*)
next
case *Cons*
from *tr* **obtain** *opt* **where**
$root' = root \vee root' = update\ (path\text{-}of\ x)\ opt\ root$
by *cases auto*
with *inv Cons* **show** *?thesis*
by (*auto simp add*: *update-eq split*: *list.splits*)

While the *Nil* case bas been mostly trivial, *Cons* requires some further care; the relevant observation from the inductive cases is exhibited via an intermediate **obtain** statement (cf. §5.3) in an *abstract* manner. Note that introduction of new existential parameters is quite typical for this kind of application, otherwise plain **have** would have been sufficient here.

qed
qed

Apparently, this pattern of initial discrimination plus abstract covering of several cases via an intermediate fact of **obtain** achieves a substantial reduction of the volume of proof text; we did not need to spell out 8 separate cases again. Even

more, the abstract characterization of the particular situation encountered here contributes to the general understanding of the main point of the proof.

In contrast, naive use of nested case-splitting from the very start could easily lead into a large number of sub-problems, which need to be all accommodated by separate proofs. In unstructured scripts such situations would be typically covered by heavy use of tactic combinators to operate on many similar goals simultaneously, with extensive tweaking of automated methods to work with these particular sub-problems uniformly. Thus we would not only lose structural clarity, but also require additional efforts in mastering automated tools (usually requiring more proof-processing time, too).

Suitable abstractions of cases need to be provided by the writer of Isar proof texts, of course. On the other hand, once that the basic setup has been given, it is usually quite easy to explore possible intermediate results interactively. This would typically proceed by inspecting the remaining *dynamic* proof state, after having issued *cases* and *auto* method invocations separately (e.g. replacing **by** proofs by a few **apply** commands temporarily).

Certainly, this phase of interactive exploration is eventually finished by turning the dynamic evolution of proof problems into static text. In the present example a large number of accidental sub-problems have been captured by simple statements within the logic, so this technique is somewhat dependent on the particular formalization of the problem and the expressive power of the underlying language. The simply-typed set-theory of Isabelle/HOL (cf. chapter 7) should reach quite far in practice, although not being unlimited.

Animation of logical objects

In the example executions of §10.5.2 we have used the existing inference engine of Isabelle to "animate" the specification of the set of file-system transitions (§10.4). The basic idea is to treat the rules from the **inductive** definition like a Prolog program, and solve emerging side-conditions by means of functional simplification, as involved in advanced Isabelle proof methods (cf. §7.3).

theorem $u \in \mathit{users} \Longrightarrow \mathit{can\text{-}exec}\ (\mathit{init}\ \mathit{users})$
  $[\mathit{Mkdir}\ u\ \mathit{perms}\ [u, \mathit{name}]]$
 apply (*rule can-exec-cons*)
 apply (*rule mkdir*)
 apply (*force simp add*: *eval*)+
 apply (*simp add*: *eval*)
 apply (*rule can-exec-nil*)
 done

Certainly, this pattern is quite far from systematic combinations of logical and functional programming techniques; the main control is left to the user by giving a suitable operational proof script.

Due to its footing on basic inference tools of Isabelle, this kind of experimen-

tal evaluation of specifications is not particularly fast. The present example takes about 0.5 s on a reasonably fast machine (see also §10.7.2). On the other hand, it is nice to have simple tools for experimentation available within the system, without demanding further efforts to maintain a link to real programming language environments. The latter would usually involve slightly awkward restrictions to strictly executable specifications. Note that the present specification includes some infinitary elements, such as unlimited sets of users and unrestricted mappings from names to sub-directories.

Despite being based on real logical inferences inside, there is no point to present this kind of symbolic evaluation as an actual Isar proof text. The proof performed at the primitive level is indeed rather accidental. Apparently, the improper proof commands **apply** and **done** (cf. §3.2.1) may have their proper use as well, even within structured formal developments of Isabelle/Isar.

Global declarations

On entering the concrete description of the "odd situation", we have introduced several declarations of **consts** and **axioms** at the global theory level (§10.6.2).

consts
users :: *uid set*
$user_1$ :: *uid*
$user_2$ :: *uid*
$name_1$ :: *name*
$name_2$ :: *name*
$name_3$ :: *name*
$perms_1$ :: *perms*
$perms_2$ :: *perms*

axioms
$user_1$*-known*: $user_1 \in users$
$user_1$*-not-root*: $user_1 \neq 0$
users-neq: $user_1 \neq user_2$
$perms_1$*-writable*: *Writable* $\in perms_1$
$perms_2$*-not-writable*: *Writable* $\notin perms_2$

This fixes a particular context for any results to be issued later on. As a purely *axiomatic extension* this certainly violates the HOL paradigm of definitionality! On the other hand, it is easy to see that the axioms are indeed satisfiable, so this extension turns out to be a conservative one (cf. §2.3), although this fact has only been established outside of the formal system.

The main intention of global declarations like this is to keep subsequent statements free from additional parameters and assumptions. So we actually did not mean to introduce an *existential* context, but a *universal* one. While it is usually accepted to fall back on plain axioms in such situations, we would

be slightly more comfortable with an explicit mechanism to manage separate contexts at the theory level succinctly.

The concept of *locales* [Kammüller *et al.*, 1999] achieves exactly this for classic Isabelle, similar to the concept of "sections" in Coq [Barras *et al.*, 1999]. Unfortunately, locales have not yet been ported to the Isabelle/Isar theory format, although this would be quite trivial. So we would morally consider the above **consts** and **axioms** as a canonical application of locales, instead of a raw axiomatic theory extension.

Local declarations in proof scripts

In the present example, we have used a few proof scripts to achieve symbolic evaluation of concrete representations of Unix file-system structures. Despite being inherently unstructured, proof scripts occasionally demand local declarations, usually to tune the behavior of proof methods to be used later on.

Due to the compositional nature of the Isar proof language, we may easily wrap up scripts into (degenerate) proof structures, in order to provide a local context for private declarations. The evaluation script for lemma *init-invariant* (§10.6.3) uses the automatic bindings of case *antecedent* and term *?thesis* to recommence the initial rule statement locally, without duplicating any text.

lemma *init-invariant*: *init users* $=$*bogus*$\Rightarrow$ *root* $\Longrightarrow$ *invariant root bogus-path*
proof −
  note *eval* = *setup access-def init-def*
  case *antecedent* **thus** *?thesis*
    ⋮

Local declarations could be just anything; here we have merely used **note** to bind facts locally. Note that in traditional Isabelle tactic scripts, such auxiliary items are usually put into the global theory context.

Advanced existential reasoning

The proof of the main invariance lemma *preserve-invariant* (§10.6.3) exhibits a number of interesting techniques, including a non-trivial pattern of reasoning along the lines of $\exists\, x.\ P\ x \Longrightarrow \exists\, x.\ Q\ x$. Existential quantification is actually hidden in the definition of *invariant root path*; using the **obtain** element of §5.3 we manage to complete the proof without ever mentioning these quantifiers explicitly.

lemma *preserve-invariant*: *root* $-x\rightarrow$ *root*$'$ $\Longrightarrow$
  invariant root path $\Longrightarrow$ *uid-of x* $=$ *user*$_1$ $\Longrightarrow$ *invariant root*$'$ *path*
proof −
  assume *tr*: *root* $-x\rightarrow$ *root*$'$
  assume *inv*: *invariant root path*
  assume *uid*: *uid-of x* $=$ *user*$_1$

Initially, we eliminate the existential content of *invariant root path*, exhibiting abstract witness elements directly to the proof text; we also decompose that inherently conjunctive statement into several individual facts (cf. §5.3).

from *inv* **obtain** *att dir* **where**
  inv_1: *access root path* $user_1$ {} = *Some* (*Env att dir*) **and**
  inv_2: *dir* ≠ *empty* **and**
  inv_3: $user_1$ ≠ *owner att* **and**
  inv_4: *access root path* $user_1$ {*Writable*} = *None*
by (*auto simp add*: *invariant-def*)

The main proof now proceeds by producing "primed" versions of $inv_1, \ldots, inv_4$ to accommodate the ultimate introduction of *invariant root′ path*. Note that $inv_3{}'$ coincides with inv_3, while $inv_4{}'$ is not labeled explicitly since it directly emerges near the very end of the proof.

⋮

from inv_3 *lookup′* **and** *not-writable* $user_1$*-not-root*
have *access root′ path* $user_1$ {*Writable*} = *None*
by (*simp add*: *access-def*)
with $inv_1{}'$ $inv_2{}'$ inv_3 **show** *?thesis* **by** (*unfold invariant-def*) *blast*

From these facts about explicit witness elements, we easily get the main result by having Isabelle's tableau prover *blast* (cf. §7.3) work out the details of conjunction and existential introduction.

qed
qed
qed

Abstraction by explicit statements

Another very common technique encountered in the proof of *preserve-invariant* (§10.6.3) is that of reducing the (conceptual and technical) complexity of the proof by inserting abstractions via explicit intermediate statements. This is certainly a rather obvious thing to do in any serious proof; albeit the established practice of unstructured tactical proving tends to proceed in one way only, decomposing statements into an increasingly large amount of "simpler" propositions emitted dynamically. By including appropriate abstract statements, the writer of Isar proof texts is enabled to keep the overall complexity under control.

We shall indicate a few notable instances of simplifications achieved by some additional Isar proof structure used together with suitable local facts.

lemma *preserve-invariant*: *root* $-x\rightarrow$ *root′* ⟹
  invariant root path ⟹ *uid-of x* = $user_1$ ⟹ *invariant root′ path*
proof −

⋮

show *?thesis*
proof *cases*

At this point we discriminate against $root' = root$. The former case achieves the main result directly. The latter case provides a useful additional assumption of *changed*: $root' \neq root$, to be used immediately to obtain an abstract view on the inductive cases of *tr*: $root\ -x\rightarrow\ root'$; several further occurrences of *changed* are encountered later on.

assume $root' = root$
with *inv* **show** *invariant root' path* **by** (*simp only*:)
next
assume *changed*: $root' \neq root$
with *tr* **obtain** *opt* **where** *root'*: $root' = update\ (path\text{-}of\ x)\ opt\ root$
by *cases auto*

⋮

Now we are about to establish the resulting invariant in a fairly trivial case, with equal access to the current path by arbitrary permissions. Claiming a local rule statement with universal parameter *perms*, we spare us to consider the concrete instances of $\{\}$ and $\{Writable\}$ separately.

have $\bigwedge perms.\ access\ root'\ path\ user_1\ perms = access\ root\ path\ user_1\ perms$
by (*simp only*: *access-update-other*)
with *inv* **show** *invariant root' path*
by (*simp only*: *invariant-def*)

⋮

Below we provide another view on the inductive cases of the fact *tr*: $root\ -x\rightarrow\ root'$. Due to the specific situation, plain **have** of a disjunctive statement is sufficient, so this is actually an example where no existential parameters are to be obtained.

with *tr path*
have $lookup\ root\ ((path\ @\ [y])\ @\ (us\ @\ [u])) \neq None \vee$
$lookup\ root\ ((path\ @\ [y])\ @\ us) \neq None$
by *cases* (*auto dest*: *access-some-lookup*)

⋮

10.7.2 Efficiency of Isabelle/Isar proof processing

Resource requirements of Isar proof processing have not been discussed so far, because this is basically not an issue for the internal bookkeeping of the Isar/VM interpreter (§3.2.3). Operations encountered here are performed quickly in practical applications, despite the theoretical complexity of higher-order unification (e.g. [Paulson, 1989]) involved in some basic proof steps of Isar (e.g. **qed** of a claim issued by **show**).

The run time of basic Isabelle/Isar applications, such as those of chapter 4, chapter 8, chapter 9, is usually limited to a few seconds for a whole theory. In interactive development, the user agent process [Proof General] typically requires slightly more time to manage the source buffer and display of prover output than the main Isabelle process.

In more complex applications (like the present one) a considerable amount of run time is spent in terminal proof steps involving automated methods, such as *blast*, *auto*, *force* (cf. §7.3). Further resources are required by advanced definitions like **inductive** and **datatype** (cf. §7.2.1). Consequently, the processing time of theories may approach the range of minutes instead of mere seconds.

In order to give a more precise account of Isabelle/Isar run time encountered in reality, we fix the following platform: AMD K7 Thunderbird CPU (900 MHz), 256 MB main memory (133 MHz), Linux 2.4.0 kernel, Poly/ML 4.1 compiler and run time system, and Isabelle99-2. All timings refer to "clean" Isabelle proof processing, *without* the quick-and-dirty mode that might get used to skip a number of internal proofs, especially those of advanced definitional packages [Berghofer and Wenzel, 1999].

An Isabelle session consists of a number of theories that are loaded in addition to the main HOL image, which already contains a collection of many basic concepts (cf. chapter 7). Our *Unix* session includes two additional theories from [Bauer *et al.*, 2001]. The overall run times are as follows.

List-Prefix	1.5 s
Nested-Environment	4.5 s
Unix	33.5 s

As long as theories from the library are static, the additional overhead of loading on demand does not really matter; such theories need to be loaded at most once during an interactive session. On the other hand, advanced applications typically consist of a number of interdependent theories in their own right. During development, the user needs to switch back and forth between different parts of that DAG structure, usually demanding frequent reloading of individual theory nodes. In fact, theory *Nested-Environment* started as a genuine part of the *Unix* session, but was moved into the generic library later on.

In any case, the Isabelle/Isar system automatically takes care to reload changed theories as required, and also ensures a consistent view on sources managed by the user agent [Proof General]. On the other hand, there is *no* specific support to avoid costly replays of individual proofs that happen to be independent of recent changes. Once that the processing time of theory nodes gets beyond a few seconds, this situation easily becomes a hindrance in development of large applications. See also the related discussion of the "Fundamental Theorem of Algebra" project [Geuvers *et al.*, 2000] performed with Coq [Barras *et al.*, 1999].

Thus the size and complexity of applications has its natural limits in the perceived performance of proof processing, both in batch mode and interactive development. The present *Unix* example is still considerably below the level of

any serious concerns, though.

We now focus on individual proofs within a single theory, namely that of *Unix*. A substantial amount of run time is actually spent in the 6 scripts of symbolic evaluation alone (examples 1–4 in §10.5.2, as well as lemma *init-invariant* in §10.6.3).

evaluation example #1	0.42 s
evaluation example #2	0.93 s
evaluation example #3	3.84 s
evaluation example #4	3.35 s
lemma *init-invariant*	2.37 s

Enormous run times for seemingly small problems are usually caused by relatively large goal states that need to be treated by simplification, involving a multitude of different cases that stem from the syntactic structure of datatype elements. The Isabelle simplifier appears to be particularly slow in conjunction with this particular kind of case-splitting. In Coq [Barras *et al.*, 1999] the highly-tuned builtin notion of $\beta\delta\iota$-reduction should perform slightly better; although Coq has a number of other performance issues beyond symbolic evaluation (cf. the experience reported in [Geuvers *et al.*, 2000]).

Most of actual Isar proof texts encountered in theory *Unix* are fairly small, with almost negligible resource requirements each; they add up to a few seconds in total, apart from the large proof of lemma *preserve-invariant* which requires 7.04 s itself. As expected, a substantial part of that time is spent in a few terminal proof steps involving automated proof methods; we indicate those contributing more than 0.5 s below.

lemma *preserve-invariant*: $root\ {-x\rightarrow}\ root' \Longrightarrow$
  $invariant\ root\ path \Longrightarrow uid\text{-}of\ x = user_1 \Longrightarrow invariant\ root'\ path$
proof −

  ⋮

  with *tr* **obtain** *opt* **where** *root'*: $root' = update\ (path\text{-}of\ x)\ opt\ root$
    by *cases auto* — (1)

    ⋮

    with *tr uid* inv_2 inv_3 *lookup changed path* **and** $user_1$*-not-root*
    have *False*
      by *cases* (*auto simp add*: *access-empty-lookup dest*: *access-some-lookup*)
        — (2)

      ⋮

      from *tr uid changed look' dir'* **obtain** *att''* **where**
          look'': $lookup\ root'\ (path\text{-}of\ x) = Some\ (Env\ att''\ dir')$

```
      by cases (auto simp add: access-empty-lookup lookup-update-some
        dest: access-some-lookup)  — (3)

  ⋮

  with tr uid inv₄ changed obtain file where
      root' = update (path-of x) (Some file) root
    by cases auto  — (4)

      ⋮

      with tr path
      have lookup root ((path @ [y]) @ (us @ [u])) ≠ None ∨
          lookup root ((path @ [y]) @ us) ≠ None
        by cases (auto dest: access-some-lookup)  — (5)

      ⋮

qed
```

We get the following distribution of run time for lemma *preserve-invariant*.

by step #1	0.51 s
by step #2	1.22 s
by step #3	1.56 s
by step #4	1.14 s
by step #5	1.15 s
remaining 146 commands	1.46 s
total	7.04 s

The proof obligations encountered in those 5 "hot-spots" indicated above are all structurally quite similar: after an initial split into the 8 inductive cases stemming from the file-system transition relation, the remaining sub-problems are solved via *auto* (cf. §7.3), which involves simplification and case-splitting over the syntactic structure of datatype elements. The latter proof problems are close to the rather slow symbolic evaluations encountered before.

Nevertheless, the overall run time behavior of the Isar proof text is fairly good. As a more detailed analysis of the remaining 1.46 s above reveals, there is virtually no penalty for the overhead of structured proof processing via the Isar/VM interpreter (cf. §3.2.3); the overall run time resources are almost completely available to the primitive inference engine below the Isar level.

This is essentially the same goal that any viable operating-system design would strive to achieve: any additional structures and policies required to provide high-level abstractions on top primitive system resources must not use up any substantial portion of these resources themselves. For example, a good operating system would typically require only 1% of CPU time for internal bookkeeping, while 99% are available to run user-space processes. Isar proof processing we

achieves a similar ratio; most of the run-time resources are left to primitive steps performed inside advanced proof methods.

Even more, the overall performance of processing well-structured Isar proof texts is usually *better* than that of unstructured Isabelle proof scripts. Typical tactic scripts consist of a large number of method invocations, operating on a single (mostly unstructured) goal state; existing subgoals get broken down to simpler (but larger) ones, to be solved eventually; new goals emerge by inheriting the context of previous ones. As a consequence, goal states arising from many individual tactic applications tend to consist of a lot of redundant information accumulated over time. Advanced proof tools need to take care of this ballast, even though most if it does not contribute to the result to be achieved.

In contrast, nicely structured Isar proof texts usually state a number of local problems that may be tackled by automated tools in isolation, indicating only those facts that are apt to contribute to the problem at hand (cf. the issue of "relevance of facts" in §7.5.2). Thus the individual proof obligations arising from the text are typically much smaller than those emerging from tactic scripts. This can make a big difference for heavily automated proof tools, such as Isabelle's *auto* or *force* (§7.3).

The problem of redundant local facts is recognized in the tradition of Isabelle tactic scripts as well; there are a number of operations to tune a goal state by removing unwanted premises. This additional tweaking is unnecessary in proper Isar texts, of course. Here one would just refrain from including unwanted facts in terminal proof steps in the first place, e.g. by restricting **from** or **with** specifications to what is really required. Thus we may gain both efficiency of proof checking and clarity of the resulting text, since irrelevant facts are excluded from opaque automated steps.

The general experience with Isabelle/Isar applications suggests an even stronger conclusion to be drawn here on the issue of efficiency of proof processing. One could argue that local steps requiring substantial amounts of run time (due to large search spaces) are somewhat questionable as an "atomic" justification in the first place — too much heavy-duty reasoning is performed out of sight of the reader. Certainly, this attitude assumes that the automated tools involved here somehow correlate the inherent complexity of a problem with real run time. Realistic proof procedures are typically quite "uneven" in that respect, though. Nevertheless, an experienced Isar proof writer would think twice about terminal steps that require excessive run time unexpectedly.

Incidently, a similar philosophy is encountered in Mizar [Rudnicki, 1992] [Trybulec, 1993] [Muzalewski, 1993] [Wiedijk, 1999], where the builtin proof procedure for finishing local proof obligations is strictly limited to a class of problems that may be decided efficiently [Rudnicki, 1987]. On the other hand, this slightly restrictive approach rules out the use of advanced tools required for a broader range of applications. Outside of the primary domain of Mizar (i.e. classical mathematics and set-theory) its applications tend to become crowded by numerous intermediate steps, to accommodate the relatively weak automation

facilities.

So it is certainly a good thing to enable Isar proof texts to incorporate arbitrarily exotic proof methods that happen to be available as tactic implementations in the raw Isabelle system. According to the general principle of liberality (cf. §1.3), it is left to the user to make proper use of what has been made available.

Chapter 11

Conclusion

11.1 Stocktaking

Taking a well-understood natural deduction framework as a starting point (chapter 2), we have introduced the versatile high-level proof language Isar (chapter 3), which supports human-readable proof texts, is generic wrt. the underlying object-logic, extensible wrt. proof tools and specific language elements, and foundationally sound by full reduction to primitive inferences. Existing approaches to mechanized theorem proving have so far covered only some of these aspects in isolation, where Isar provides a coherent view of the whole picture. This broad scope of Isar marks a distinctive advantage, consequently we have been able to cover rather general techniques for structured proof composition (chapter 5 and chapter 6).

The Isar concepts have turned out sufficiently simple and mature to provide a viable basis for the robust system implementation Isabelle/Isar [Wenzel, 2001a]. We have also been able to demonstrate that the generic framework may be actually instantiated to the concrete setting of Isabelle/HOL (chapter 7). Practical usability has been demonstrated both for the generic framework (chapter 4 and chapter 8) and the Isabelle/HOL instantiation (chapter 9 and chapter 10). In particular, we have been able to include *complete* formal proof texts of meaningful examples (despite the extra printed pages demanded here).

By having achieved a new quality of intelligible semi-automated reasoning, we expect to address new application areas, as well as new users who have not considered interactive theorem proving as something reasonable so far. Note that Isar does not necessarily attract exactly the same kind of users as tactical theorem proving. Due to the very shift of proof development paradigms there are quite different techniques required by proof writers. Experts in the old tactical style certainly do have to unlearn some of their habits to master structured proof composition in Isar, where fresh users would typically have fewer problems.

Concerning our main objective of human-readable proofs, we observe that there

is no single underlying principle of intelligible texts. Drawing from a certain repertoire of common elements, proof authors need to spend some care on composing an adequate record of formal reasoning. From the Isar perspective the following aspects have turned out particular important.

- Moderate inclusion of explicit propositions in the text.

  Proof authors certainly do have to state key propositions explicitly, but need to avoid clutter due to excessive coverage of concrete terms. Isar provides specific support for casual term abbreviations via higher-order matching (§3.2.3 and §3.4.1).

- Clear indication of the present role of logical entities (assumptions versus conclusions, universal versus existential parameters etc.).

  Isar provides a particularly rich collection of specific context elements (§3.2.3, §3.3.1, and §5.3). Appropriate variations on conclusions are available as well (§3.2.3 and §3.3.3). All of this is reduced to raw $\bigwedge/\Longrightarrow$ statements internally.

- Succinct references to previous facts, while avoiding explicit labels.

  Isar's **then** element (§3.2.3) provides the most fundamental mechanism of referring directly to the preceding result. Several derived elements exploit this principle further (§3.3.3), even at a somewhat larger scale within the calculational proof style (§6.3 and §6.4.3). Referencing facts explicitly should be really restricted to rare situations of more complex dependencies (e.g. multiple uses).

- Indication of relevance of facts involved in particular proof steps.

  Chaining of facts via **then** (or its derived forms) enables natural techniques of "feeding" results into consecutive goals. For general (initial) proof steps of single rules, this gives rise to mixed forward-backward reasoning (§5.2.3). In conjunction with automated methods, the Isar text may record the contributing collection of relevant facts explicitly, which improves both readability and scalability of automated means (§7.5.2).

We have also gained further insights into the role of some general aspects of theorem proving in the particular context of human-readable proofs.

- Readability requires a clear separation of static proof text from dynamic goal state.

  Tactic languages are apt to let arbitrary elements from implicit goal configurations intrude the recorded source, e.g. names of local parameters introduced implicitly by previous refinements. The Isar proof processor refrains from inspecting the internal structure of facts or goals at all. Direct transformation of (internal) goal states have been discontinued. Proof writers need to "answer" a particular form of subgoal by explicit text, with their own choice of parameters (§3.2.3 and §5.2.1).

- The importance of extra-logical concepts.

  Achieving a high-level view on formal logic does not necessarily involve new (exotic) calculi. Instead, we have built a different extra-logical layer of structured proof configurations for the Isar interpretation process (§3.2.3). Based on this rich auxiliary structure, the proof interpreter "drives" a few logical primitives (§2.2 and §2.4); the example interpretation trace at the end of §3.2.3 illustrates the relationship of the two layers particularly well.

  Note that some Isar concepts do not have any logical impact at all, e.g. term abbreviations (§3.2.3 and §3.4.1). Here we exploit the powerful mechanism of higher-order unification for abstract abbreviation patterns, even with Hindley-Milner polymorphism (§3.4.3), without having to bother about extending the logical foundations.

- Automated reasoning techniques are *not* a key issue.

  Big-step reasoning with automated bridging of considerable gaps in the course of formal reasoning has often been proposed as the standard way to achieve "high-level" proofs of some form.

  Isar demonstrates that careful structural arrangement of proof elements may greatly reduce the need for automated tools. The very core mechanisms of Isar proof composition already achieve decent arrangements, mainly by means of higher-order resolution of single natural deduction rules (which may involve higher-order unification). Furthermore, indicating relevant facts in (local) automated steps may reduce the complexity of proof problems considerably, which enables to get farther by simpler proof tools (e.g. plain rewriting).

11.2 Future work

Despite the achievements of the present work on Isabelle/Isar, this can only be another stepping-stone towards further investigations on high-level formal proof languages at large. Apart from various technical details of Isar proof processing there are also some issues of putting the Isar concepts into a wider context.

Synthesized results

Isar takes the existing natural deduction framework of Isabelle/Pure as a starting point, exploiting much of its inherent potential for the purposes of structured proof processing, rather than mere tactical theorem proving. On the other hand, Isar ignores Isabelle's capability of schematic goal statements, which would admit incremental synthesis of proven results by stepwise refinement (similar to logic-programming techniques), e.g. see [Paulson, 2001a].

Isar proof texts really need to be fully specified in advance. Whereas unbound schematic variables may well occur inside of *internal* goal states (after some initial refinement), the consecutive proof body needs to accommodate this again by definite text elements (typically involving **fix**/**assume** and **show**). Excluding schematic statements is a tribute to overall readability, i.e. written texts may not just "mutate" dynamically. Moreover, schematic goals would violate modularity of sub-proofs, since the course of reasoning in the body affects the result instantiation achieved eventually. (Modularity is a key factor to support large-scale applications.)

Nevertheless, the general idea of synthesizing results is not completely alien to structured reasoning, although one might have to rethink existing tactical approaches in terms of Isar concepts. In particular, the calculational proof style (chapter 6) offers a general framework for synthesizing facts in a forward fashion, by consecutive composition of chains of intermediate results.

Based on this observation, we have already experimented with some mechanisms for stepwise synthesis of verified programs in Hoare Logic elsewhere [Wenzel, 2001c, §13–14]. Here the structure of the resulting Isar text corresponds nicely to that of the program. On the other hand, practical usability demands further refinements of these ideas, such as proper "export" of finished fragments without repeated statements of Hoare triples in the text, and more handsome right-to-left development to accommodate incremental reasoning from post-conditions to pre-conditions.

Unusual logics

The Isar proof language provides a faithful high-level view of the underlying natural deduction framework of Isabelle/Pure. Object-logics that directly conform to the induced notion of statements over meta-level $\bigwedge/\Longrightarrow$ connectives may immediately benefit from the Isar layer (e.g. chapter 4 and chapter 8).

On the other hand, "unusual" logics (from the natural deduction perspective) may demand different encodings within the Isabelle framework to begin with. For example, existing formulations of modal, temporal, and linear logics depend on a version of sequent calculus [Isabelle library], featuring explicit representations of complete proof configurations as complex meta-logical judgments. Unfortunately, that slightly indirect view on the object-logic results in impractical Isar proof texts, requiring statements of whole sequents in the text over and over again; partially specified configurations with schematic sequent variables are also unavailable. So the Isar text would basically degenerate into a low-level trace of sequent-calculus proofs. Workable interactive development would also pose a fundamental problem.

Unusual logics are rarely used in Isabelle at all, so one might argue that even the traditional tactical view turns out as slightly inconvenient here. Certainly, one might consider to augment the underlying framework to cover modalities itself, notably a meta-level "$\Box$" operator. Then the Isar language might be

extended accordingly to achieve reasonable representations of modal reasoning. There are also alternative approaches [Basin and Matthews, 2001] of "encoding less well behaved logics" directly within a pure natural deduction framework, namely via "labeled deductive systems". It would be interesting to see if this provides a viable starting point for adequate proof texts within Isar as well. That view could become particularly relevant in practice to support similar semantic embeddings of modal logics within the existing natural deduction environment of Isabelle/HOL.

Beyond linear proof processing

The Isar language is based on left-to-right interpretation of individual proof commands, operating on a structured proof configuration inside. This paradigm induces a canonical sequential reading of the static proof text. Some minor drawbacks of the left-to-right bias may be encountered both at small and large scale, such as occasional "inversions" of the wording (e.g. patterns of the form "**{** ... **}** **note** $a = \mathit{this}$" in §5.2.3), or the limitations of non-linear forward proof patterns (cf. the discussion of "generalized case-splitting" in §5.5.3).

Further conveniences not available in the present incremental interpretation model are global static analysis (notably simultaneous type-checking, cf. §3.4.3), and cumulative error reporting of failed proof steps (Isabelle/Isar currently stops at the first problem encountered). Batch-mode proof processing (where the whole text is available at once) could achieve such features quite easily, but we certainly do not intend to trade the virtues of incremental proof development (including interactive experimentation) for such relatively marginal issues.

In fact, the potential to support block-structured top-down development is already present in Isabelle/Isar, since sub-proofs may be processed independently. It is already possible to skip failed proof steps by inserting a "dummy proof" temporarily [Wenzel, 2001a]. Further convenience is mainly a matter of user-interface support; the existing Proof General technology [Aspinall, 2000] appears to have sufficient potential to overcome its present focus on linear proof script processing at a later stage.

Large-scale theory development

As may be learned from the Mizar project [Rudnicki, 1992] [Trybulec, 1993], both readable proofs and viable support for modular theory concepts (with mathematical structures) are important prerequisites for large-scale library developments, attracting contributions by many authors.

Isar already inherits a canonical concept of derived natural deduction rules from the underlying framework. This essentially admits to abstract certain reasoning patterns into a meta-level theorem, which may be used as a single proof rule later on. So we already achieve small-scale packaging of recurrent proof schemes, analogous to functional abstraction.

Integration of the Isar language with actual module systems for logical environments would be certainly desirable as well. Here the existing concept of "locales" [Kammüller *et al.*, 1999] for Isabelle/Pure appears to be particularly promising, say by generalizing its immediate view of $\bigwedge/\Longrightarrow$ contexts to that of Isar proof contexts. Such a version of locales would enable packaging of Isar elements **fix**, **assume**, **def**, **let**, **note** etc., maybe even **obtain**. Further issues may arise when moving between different contexts within structured proofs.

Meta-theoretic studies of Isar

Obviously the interpretation process of Isar commands (§3.2.3), which provides an operational semantics, may be exploited for further meta-theoretical studies of the language. The standard repertoire includes suitable notions of correctness and completeness, in terms of primitive inferences of the underlying framework. It would be particularly interesting to formalize these aspects of Isar within Isabelle/Isar itself (e.g. based on an inductive model in Isabelle/HOL). This would continue to the old tradition of presenting new programming language designs by giving an interpreter within the same language.

On the other hand, even such fully formal treatment of Isar meta-theory would be of relatively little practical relevance, apart from providing another concrete application. In practice, correctness of Isar proof processing is better achieved analogously to Milner's "Correctness by Construction", which means here that high-level proof elements are fully reduced to primitive inferences at run-time. Recall that Isar even treats the primitive level as fully abstract (§1.4), being independent of the exact internal structure of the results.

Moreover, completeness means for practical purposes that a reasonable range of applications may be addressed. This has already been demonstrated empirically, by the reference examples of chapter 8, chapter 9, and chapter 10. In fact, these "live demonstrations" did not yet stretch the Isabelle/Isar environment too far. Further applications have emerged in the meantime (also by other people), and even more may be expected for the future.

Aliquantum iam a locutione cessandum est,
ut si ad aliorum miracula enarranda tendimus,
loquendi vires interim per silentium reparemus.

Bibliography

[Abel *et al.*, 2001] A. Abel, B.-Y. E. Chang, and F. Pfenning. Human-readable machine-verifiable proofs for teaching constructive logic. IJCAR Workshop on Proof Transformations, Proof Presentations and Complexity of Proofs (PTP-01), http://www.tcs.informatik.uni-muenchen.de/~abel/ptp01.ps.gz, 2001.

[Agda] Agda homepage. http://www.cs.chalmers.se/~catarina/agda/.

[Andrews, 1986] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.

[Arkoudas, 1998] K. Arkoudas. Deduction vis-a-vis computation: The need for a formal language for proof engineering. Unpublished paper, The MIT Express project, http://www.ai.mit.edu/projects/express/, June 1998.

[Asperti *et al.*, 2001] A. Asperti, L. Padovani, C. S. Coen, and I. Schena. HELM and the semantic math-web. In Boulton and Jackson [2001].

[Aspinall, 2000] D. Aspinall. Proof General: A generic tool for proof development. In *European Joint Conferences on Theory and Practice of Software (ETAPS)*, 2000.

[Back and von Wright, 1999] R. J. Back and J. von Wright. Structured derivations: A method for doing high-school mathematics carefully. Technical Report 246, Turku Centre for Computer Science, 1999.

[Back *et al.*, 1997] R. J. Back, J. Grundy, and J. von Wright. Structured calculational proof. *Formal Aspects of Computing*, 9:469–483, 1997.

[Balaa and Bertot, 2000] A. Balaa and Y. Bertot. Fix-point equations for well-founded recursion in type theory. In Harrison and Aagaard [2000].

[Bali] The Bali project. http://isabelle.in.tum.de/Bali/.

[Bancerek and Carlson, 1993] G. Bancerek and P. Carlson. Mizar and the machine translation of mathematics documents. Unpublished paper, 1993.

[Barendregt and Geuvers, 2001] H. Barendregt and H. Geuvers. Proof assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2001.

[Barendregt *et al.*, 1995] H. P. Barendregt, G. Barthe, and M. Ruys. A two level approach towards lean proof-checking. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs (TYPES'95)*, volume 1158 of *LNCS*, 1995.

[Barendregt, 1992] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 118–309. Oxford University Press, 1992.

[Barras *et al.*, 1999] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual, version 6.3*. INRIA, 1999.

[Barwise and Etchemendy, 1995] J. Barwise and J. Etchemendy. Hyperproof. CSLI Lecture Notes, Stanford, 1995. http://www-csli.stanford.edu/hp/.

[Barwise and Etchemendy, 1998] J. Barwise and J. Etchemendy. Computers, visualization, and the nature of reasoning. In T. W. Bynum and J. H. Moor, editors, *The Digital Phoenix: How Computers are Changing Philosophy*. London: Blackwell, 1998.

[Basin and Matthews, 2001] D. Basin and S. Matthews. Logical frameworks. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, second edition*. Reidel, 2001.

[Bauer and Wenzel, 2000] G. Bauer and M. Wenzel. Computer-assisted mathematics at work — the Hahn-Banach theorem in Isabelle/Isar. In T. Coquand, P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs: TYPES'99*, volume 1956 of *LNCS*, 2000.

[Bauer and Wenzel, 2001] G. Bauer and M. Wenzel. Calculational reasoning revisited — an Isabelle/Isar experience. In Boulton and Jackson [2001].

[Bauer *et al.*, 2001] G. Bauer, T. Nipkow, L. C. Paulson, T. M. Rasmussen, and M. Wenzel. The supplemental Isabelle/HOL library. Part of the Isabelle99-2 distribution, http://isabelle.in.tum.de/library/HOL/Library/document.pdf, 2001.

[Bauer, 1999] G. Bauer. Lesbare formale Beweise in Isabelle/Isar — der Satz von Hahn-Banach. Master's thesis, TU München, November 1999. http://home.in.tum.de/~bauerg/HahnBanach-DA.pdf.

[Bauer, 2001a] G. Bauer. The Hahn-Banach Theorem for real vector spaces. Part of the Isabelle99-2 distribution, http://isabelle.in.tum.de/library/HOL/HOL-Real/HahnBanach/document.pdf, February 2001.

[Bauer, 2001b] G. Bauer. Some properties of CTL. http://isabelle.in.tum.de/library/HOL/CTL/document.pdf, June 2001.

[Benl *et al.*, 1998] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the Minlog system. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume II: Systems and Implementation Techniques of *Applied Logic Series*. Kluwer Academic Publishers, 1998.

[Berghofer and Nipkow, 2000] S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In Harrison and Aagaard [2000].

[Berghofer and Wenzel, 1999] S. Berghofer and M. Wenzel. Inductive datatypes in HOL — lessons learned in Formal-Logic Engineering. In Bertot et al. [1999].

[Berghofer and Wenzel, 2001] S. Berghofer and M. Wenzel. *The Isabelle System Manual*, 2001. Part of the Isabelle99-2 distribution, http://isabelle.in.tum.de/doc/system.pdf.

[Bertot and Thery, 1996] Y. Bertot and L. Thery. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 11, 1996.

[Bertot *et al.*, 1997] Y. Bertot, T. Kleymann-Schreiber, and D. Sequeira. Implementing proof by pointing without a structure editor. Technical report, LFCS Edinburgh, 1997.

[Bertot *et al.*, 1999] Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors. *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *LNCS*, 1999.

[Boulton and Jackson, 2001] R. J. Boulton and P. B. Jackson, editors. *Theorem Proving in Higher Order Logics: TPHOLs 2001*, volume 2152 of *LNCS*, 2001.

[Burstall, 1998] R. Burstall. Teaching people to write proofs: a tool. In *CafeOBJ Symposium, Numazu, Japan*, April 1998.

[Church, 1940] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, pages 56–68, 1940.

[Cohn, 1995] A. Cohn. Proof accounts in HOL. Unpublished paper, http://www.cl.cam.ac.uk/~mjcg/papers/AvraAccountsPaper.ps.gz, 1995.

[Constable *et al.*, 1986] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[Coquand and Coquand, 1999] T. Coquand and C. Coquand. Structered type theory. Workshop on Logical Frameworks and Meta-Languages, Paris, France, 1999.

[Coquand and Paulin-Mohring, 1990] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *LNCS*, 1990.

[Coscoy *et al.*, 1995] Y. Coscoy, G. Kahn, and L. Théry. Extracting text from proofs. In *Typed Lambda Calculus and Applications*, volume 902 of *LNCS*. Springer, 1995.

[Dahn and Wolf, 1994] B. I. Dahn and A. Wolf. A calculus supporting structured proofs. *Journal of Information Processing and Cybernetics (EIK)*, 30(5-6):261–276, 1994. Akademie Verlag Berlin.

[Dahn *et al.*, 1997] B. I. Dahn, J. Gehne, T. Honigmann, and A. Wolf. Integration of automated and interactive theorem proving in ILF. In W. McCune, editor, *14th International Conference on Automated Deduction — CADE-14*, volume 1249 of *LNAI*. Springer, 1997.

[Davey and Priestley, 1990] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[de Bruijn, 1980] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic*, pages 579–606. Academic Press, 1980.

[Despeyroux *et al.*, 1997] J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, volume 1210 of *LNCS*. Springer, 1997.

[Dijkstra and Scholten, 1990] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and monographs in computer science. Springer, 1990.

[Farmer *et al.*, 1993] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, Oct 1993.

[Gentzen, 1935] G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 1935.

[Geuvers *et al.*, 2000] H. Geuvers, F. Wiedijk, J. Zwanenburg, R. Pollack, and H. Barendregt. The "Fundamental Theorem of Algebra" project, 2000. http://www.cs.kun.nl/~freek/fta/index.html.

[Gordon and Melham, 1993] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Gordon *et al.*, 1979] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.

[Gordon, 1985a] M. J. C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, 1985.

[Gordon, 1985b] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. Technical Report 77, University of Cambridge Computer Laboratory, 1985.

[Gordon, 1988] M. J. C. Gordon. HOL: a proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1988.

[Gordon, 2000] M. J. C. Gordon. From LCF to HOL: a short history. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000. http://www.cl.cam.ac.uk/~mjcg/papers/HolHistory.html.

[Grundy and Newey, 1998] J. Grundy and M. Newey, editors. *Theorem Proving in Higher Order Logics: TPHOLs '98*, volume 1479 of *LNCS*, 1998.

[Grundy, 1991] J. Grundy. Window inference in the HOL system. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the International Workshop on the HOL Theorem Proving System and Its Applications*. ACM SIGDA, IEEE Computer Society Press, 1991.

[Gunter and Felty, 1997] E. L. Gunter and A. Felty, editors. *Theorem Proving in Higher Order Logics: TPHOLs '97*, volume 1275 of *LNCS*, 1997.

[Hallgren and Ranta, 2000] T. Hallgren and A. Ranta. An extensible proof text editor. In *Logic for Programming and Automated Reasoning (LPAR 2000)*, volume 1955 of *LNAI*. Springer, 2000.

[Harrison and Aagaard, 2000] J. Harrison and M. Aagaard, editors. *Theorem Proving in Higher Order Logics: TPHOLs 2000*, volume 1869 of *LNCS*, 2000.

[Harrison, 1995] J. Harrison. Inductive definitions: automation and application. In P. J. Windley, T. Schubert, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 8th International Workshop*, volume 971 of *LNCS*, pages 200–213, Aspen Grove, Utah, 1995. Springer.

[Harrison, 1996a] J. Harrison. HOL done right. Unpublished paper, 1996.

[Harrison, 1996b] J. Harrison. A Mizar mode for HOL. In Wright et al. [1996], pages 203–220.

[Harrison, 1996c] J. Harrison. *Theorem proving with the real numbers*. PhD thesis, University of Cambridge Computer Laboratory, 1996. Technical Report number 408, http://www.ftp.cl.cam.ac.uk/ftp/papers/reports/TR408-jrh-Theorem-Proving-with-the-Real-Numbers.ps.gz.

[Harrison, 1998] J. Harrison. Formalizing Dijkstra. In Grundy and Newey [1998].

[Henkin, 1950] L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, 1950.

[Heuser, 1986] H. Heuser. *Funktionalanalysis: Theorie und Anwendung*. Teubner, 1986.

[Hindley, 1969] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146, 1969.

[Hofmann, 1999] M. Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual IEEE Symposium on Logic in Computer Science (LICS'99)*, volume 158. IEEE Computer Society, 1999.

[Hofstadter, 1979] D. R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books, New York, 1979.

[Hoover and Rudnicki, 1996] H. J. Hoover and P. Rudnicki. Teaching freshman logic with Mizar-MSE. DIMACS Workshop on Teaching Logic and Reasoning in an Illogical World, http://web.cs.ualberta.ca/~hoover/dimacs-teaching-logic/paper.html, 1996.

[Hutter, 2000] D. Hutter. Management of change in structured verification. In *Proceedings Automated Software Engineering (ASE-2000)*. IEEE, 2000. http://www.dfki.de/vse/papers/hutter00.ps.gz.

[Isabelle library] Isabelle theory library. http://isabelle.in.tum.de/library/.

[Jape] Jape — a framework for building interactive proof editors. http://users.comlab.ox.ac.uk/bernard.sufrin/jape.html.

[Kammüller *et al.*, 1999] F. Kammüller, M. Wenzel, and L. C. Paulson. Locales: A sectioning concept for Isabelle. In Bertot et al. [1999].

[Kaufmann *et al.*, 2000] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer, 2000.

[Klein *et al.*, 2001] G. Klein, T. Nipkow, D. v. Oheimb, and C. Pusch. μJava. Part of the Isabelle99-2 distribution, http://isabelle.in.tum.de/library/HOL/MicroJava/document.pdf, 2001.

[Lamport and Paulson, 1999] L. Lamport and L. C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, 1999.

[Lamport, 1994] L. Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, 1994.

[McAllester, 1988] D. A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics*. MIT Press, 1988.

[McAllester, 1990] D. A. McAllester. Automatic recognition of tractability in inference relations. Technical Report 1215, MIT, 1990.

[McCarthy, 1960] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine (part I). *Communications of the ACM*, April 1960. http://www-formal.stanford.edu/jmc/recursive.html.

[Miller, 1991] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4), 1991. http://www.cse.psu.edu/~dale/papers/jlc91.pdf.

[Milner, 1978] R. Milner. A theory of type polymorphism in programming. *J. Comp. Sys. Sci.*, 17, 1978.

[Mizar library] Mizar mathematical library. http://www.mizar.org/library/.

[Mizar MSE] Mizar MSE. http://www.cs.ualberta.ca/~piotr/Mizar-MSE/.

[Müller and Slind, 1997] O. Müller and K. Slind. Treating partiality in a logic of total functions. *The Computer Journal*, 40(10), 1997.

[Müller *et al.*, 1999] O. Müller, T. Nipkow, D. v. Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9, 1999.

[Muzalewski, 1993] M. Muzalewski. *An Outline of PC Mizar*. Fondation of Logic, Mathematics and Informatics — Mizar Users Group, 1993. http://www.cs.kun.nl/~freek/mizar/mizarmanual.ps.gz.

[Naraschewski and Wenzel, 1998] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in Higher-Order Logic. In Grundy and Newey [1998].

[Naraschewski, 2001] W. Naraschewski. *Teams as Types — A Formal Treatment of Authorisation in Groupware*. PhD thesis, TU München, 2001.

[Nederpelt *et al.*, 1994] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer, editors. *Selected Papers on Automath*, Studies in Logic 133. North Holland, 1994.

[Nipkow and Paulson, 2001] T. Nipkow and L. C. Paulson. *Isabelle/HOL — The Tutorial*, 2001. Part of the Isabelle99-2 distribution, http://isabelle.in.tum.de/doc/tutorial.pdf.

[Nipkow and Prehofer, 1993] T. Nipkow and C. Prehofer. Type checking type classes. In *20th ACM Symp. Principles of Programming Languages*, 1993.

[Nipkow *et al.*, 2001] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle's Logics: HOL*, 2001. Part of the Isabelle99-2 distribution, http://isabelle.in.tum.de/doc/logics-HOL.pdf.

[Nipkow, 1993] T. Nipkow. Order-sorted polymorphism in Isabelle. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.

[Oheimb, 2001] D. v. Oheimb. *Analyzing Java in Isabelle/HOL — Formalization, Type Safety and Hoare Logic*. PhD thesis, TU München, 2001. http://www4.in.tum.de/~oheimb/diss/.

[Owre and Shankar, 1997] S. Owre and N. Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, 1997.

[Owre *et al.*, 1996] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *LNCS*, 1996.

[Paulin-Mohring, 1993] C. Paulin-Mohring. Inductive definitions in the system Coq — rules and properties. In *Proceedings of Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, 1993.

[Paulson and Nipkow, 1994] L. C. Paulson and T. Nipkow. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[Paulson, 1986] L. C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3, 1986. Revised version: http://www.cl.cam.ac.uk/Research/Reports/TR082-lcp-higher-order-resolution.pdf.

[Paulson, 1989] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[Paulson, 1990] L. C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990.

[Paulson, 1991] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[Paulson, 1993] L. C. Paulson. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning*, 11(3), 1993.

[Paulson, 1994] L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *12th International Conference on Automated Deduction — CADE-12*, volume 814 of *LNAI*, pages 148–161. Springer, 1994.

[Paulson, 1995] L. C. Paulson. Set theory for verification: II. Induction and recursion. *Journal of Automated Reasoning*, 15(2), 1995.

[Paulson, 1997] L. C. Paulson. Generic automatic proof tools. In R. Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*. MIT Press, 1997.

[Paulson, 1999] L. C. Paulson. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science*, 5(3), 1999.

[Paulson, 2001a] L. C. Paulson. *Introduction to Isabelle*, 2001. Part of the Isabelle99-2 distribution, http://isabelle.in.tum.de/doc/intro.pdf.

[Paulson, 2001b] L. C. Paulson. *The Isabelle Reference Manual*, 2001. Part of the Isabelle99-2 distribution, http://isabelle.in.tum.de/doc/ref.pdf.

[PDP Unix Preservation Society] PDP Unix preservation society home page. http://minnie.cs.adfa.edu.au/PUPS/.

[Pfenning and Elliott, 1988] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, 1988.

[Pfenning and Paulin-Mohring, 1990] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, volume 442 of *LNCS*, 1990.

[Pitts, 1993] A. Pitts. The HOL logic. In Gordon and Melham [1993], pages 191–232.

[Pollack, 2000] R. Pollack. Dependently typed records for representing mathematical structure. In Harrison and Aagaard [2000].

[Prazmowski and Rudnicki, 1993] K. Prazmowski and P. Rudnicki. Mizar-MSE primer. Unpublished paper, http://ugweb.cs.ualberta.ca/~c272/Online/Primer.html, 1993.

[Proof General] Proof General — Organize your proofs! http://www.proofgeneral.org/home/proofgen/.

[Regensburger, 1995] F. Regensburger. HOLCF: Higher order logic of computable functions. In E. Schubert, P. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 971 of *LNCS*, 1995.

[Reif, 1992] W. Reif. The KIV-system: Systematic construction of verified software. In D. Kapur, editor, *11th International Conference on Automated Deduction — CADE-11*, volume 607 of *LNAI*. Springer, 1992.

[Ritchie and Thompson, 1974] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *C. ACM*, 1974. http://cm.bell-labs.com/cm/cs/who/dmr/cacm.html.

[Rudnicki, 1987] P. Rudnicki. Obvious inferences. *Journal of Automated Reasoning*, 3, 1987.

[Rudnicki, 1992] P. Rudnicki. An overview of the MIZAR project. In *1992 Workshop on Types for Proofs and Programs*. Chalmers University of Technology, Bastad, 1992.

[Ruys, 1999] M. Ruys. *Studies in Mechanical Verification of Mathematical Proofs*. PhD thesis, KU Nijmegen, 1999.

[Schroeder-Heister, 1984] P. Schroeder-Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49(4), 1984.

[Simons, 1996] M. Simons. *The Presentation of Formal Proofs*. PhD thesis, Technische Universität Berlin, 1996.

[Simons, 1997] M. Simons. Proof presentation for Isabelle. In Gunter and Felty [1997].

[Slind, 1996] K. Slind. Function definition in higher order logic. In Wright et al. [1996].

[Slind, 1997] K. Slind. Derivation and use of induction schemes in higher-order logic. In Gunter and Felty [1997].

[Slotosch, 1997] O. Slotosch. Higher order quotients and their implementation in Isabelle HOL. In Gunter and Felty [1997].

[Syme, 1997a] D. Syme. DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, 1997.

[Syme, 1997b] D. Syme. Proving Java type soundness. Technical Report 427, University of Cambridge Computer Laboratory, 1997.

[Syme, 1998] D. Syme. *Declarative Theorem Proving for Operational Semantics*. PhD thesis, University of Cambridge, 1998.

[Syme, 1999] D. Syme. Three tactic theorem proving. In Bertot et al. [1999].

[Tanenbaum, 1992] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[Thompson, 1991] S. Thompson. *Type theory and functional programming*. Addison-Wesley, 1991.

[Torvalds and others] L. Torvalds et al. The Linux kernel archives. http://www.kernel.org.

[Trybulec, 1993] A. Trybulec. Some features of the Mizar language. Presented at a workshop in Turin, Italy, 1993.

[Tutch] Tutorial proof checker. http://www.tcs.informatik.uni-muenchen.de/~abel/tutch/.

[Unix Heritage Society] The Unix heritage society. http://minnie.cs.adfa.edu.au/TUHS/.

[Verhoeven and Backhouse, 1999] R. Verhoeven and R. Backhouse. Interfacing program construction and verification. In J. Wing and J. Woodcock, editors, *FM99: The World Congress in Formal Methods*, volume 1708 and 1709 of *LNCS*, 1999.

[Wenzel, 1994] M. Wenzel. Axiomatische Typ-Klassen in Isabelle. Master's thesis, TU München, 1994.

[Wenzel, 1997] M. Wenzel. Type classes and overloading in higher-order logic. In Gunter and Felty [1997].

[Wenzel, 1999] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Bertot et al. [1999].

[Wenzel, 2001a] M. Wenzel. *The Isabelle/Isar Reference Manual*, 2001. Part of the Isabelle99-2 distribution, http://isabelle.in.tum.de/doc/isar-ref.pdf.

[Wenzel, 2001b] M. Wenzel. Lattices and orders in Isabelle/HOL. Part of the Isabelle99-2 distribution, http://isabelle.in.tum.de/library/HOL/Lattice/document.pdf, February 2001.

[Wenzel, 2001c] M. Wenzel. Miscellaneous Isabelle/Isar examples for higher-order logic. Part of the Isabelle99-2 distribution, http://isabelle.in.tum.de/library/HOL/Isar_examples/document.pdf, February 2001.

[Wenzel, 2001d] M. Wenzel. Some aspects of Unix file-system security. Part of the Isabelle99-2 distribution, http://isabelle.in.tum.de/library/HOL/Unix/document.pdf, February 2001.

[Wenzel, 2001e] M. Wenzel. *Using Axiomatic Type Classes in Isabelle*, 2001. Part of the Isabelle99-2 distribution, http://isabelle.in.tum.de/doc/axclass.pdf.

[Wiedijk, 1999] F. Wiedijk. Mizar: An impression. Unpublished paper, 1999. http://www.cs.kun.nl/~freek/mizar/mizarintro.ps.gz.

[Wiedijk, 2000] F. Wiedijk. The mathematical vernacular. Unpublished paper, 2000. http://www.cs.kun.nl/~freek/notes/mv.ps.gz.

[Wiedijk, 2001a] F. Wiedijk. Digital math WWW page, 2001. http://www.cs.kun.nl/~freek/digimath/.

[Wiedijk, 2001b] F. Wiedijk. Mizar light for HOL light. In Boulton and Jackson [2001].

[Wright *et al.*, 1996] J. Wright, J. Grundy, and J. Harrison, editors. *Theorem Proving in Higher Order Logics: TPHOLs '96*, volume 1125 of *LNCS*, 1996.

[Zammit, 1999a] V. Zammit. On the implementation of an extensible declarative proof language. In Bertot et al. [1999].

[Zammit, 1999b] V. Zammit. *On the Readability of Machine Checkable Formal Proofs*. PhD thesis, University of Kent, 1999.

Index