Isar — a Generic Interpretative Approach to Readable Formal Proof Documents

Markus Wenzel*

Technische Universität München
Institut für Informatik, Arcisstraße 21, 80290 München, Germany
http://www.in.tum.de/~wenzelm/

Abstract. We present a generic approach to readable formal proof documents, called *Intelligible semi-automated reasoning* (*Isar*). It addresses the major problem of existing interactive theorem proving systems that there is no appropriate notion of proof available that is suitable for human communication, or even just maintenance. Isar's main aspect is its formal language for natural deduction proofs, which sets out to bridge the semantic gap between internal notions of proof given by state-of-the-art interactive theorem proving systems and an appropriate level of abstraction for user-level work. The Isar language is both human readable and machine-checkable, by virtue of the Isar/VM interpreter.
Compared to existing declarative theorem proving systems, Isar avoids several shortcomings: it is based on a few basic principles only, it is quite independent of the underlying logic, and supports a broad range of automated proof methods. Interactive proof development is supported as well. Most of the Isar concepts have already been implemented within Isabelle. The resulting system already accommodates simple applications.

1 Introduction

Interactive theorem proving systems such as HOL [10], Coq [7], PVS [15], and Isabelle [16], have reached a reasonable level of maturity in recent years. On the one hand supporting expressive logics like set theory or type theory, on the other hand having acquired decent automated proof support, such systems provide quite powerful environments for sizeable applications. Taking Isabelle/HOL as an arbitrary representative of these *semi-automated reasoning* systems, typical applications are the formalization of substantial parts of the Java type system and operational semantics [14], formalization of the first 100 pages of a semantics textbook [13], or formal proof of Church-Rosser property of λ-reductions [12].

Despite this success in actually formalizing parts of mathematics and computer science, there are still obstacles in addressing a broad range of users. One of the main problems is that, paradoxically, none of the major semi-automated reasoning systems support an adequate *primary* notion of proof that is amenable to human understanding. Typical prover input languages are rather arcane, demanding a steep learning curve of users to write any proof scripts at all. Even

* Research supported by DFG grant Ni 491/2-1.

worse, the resulting texts are very difficult to understand, usually requiring stepwise replay in the system to make anything out of it. This situation is bad enough for proof maintenance, but is impossible for communicating formal proofs — the fruits of the formalization effort — to a wider audience.

According to folklore, performing proof is similar to programming. Comparing current formal proof technology with that of programming languages and methodologies, though, we seem to be stuck at the assembly language level. There are many attempts to solve this problem, like providing user interfaces for theorem provers that help users to put together proof scripts, or browser tools presenting the prover's internal structures, or generators and translators to convert between different notions of proof — even natural language.

The Mizar System [19, 22] pioneered a different approach, taking the issue of a human-readably *proof language* seriously. More recently, several efforts have been undertaken to transfer ideas of Mizar into the established tradition of tactical theorem proving, while trying to avoid its well-known shortcomings. The DECLARE system [20, 21] has been probably the most elaborate, so far.

Our approach, which is called *Intelligible semi-automated reasoning* (*Isar*), can be best understood in that tradition, too. We aim to address the problem at its very core, namely the primary notion of formal proof that the systems offers to its users, authors and audience alike. Just as the programming community did some decades ago, we set out to develop a high-level formal language for proofs that is designed with the human in mind, rather than the machine.

The Isar language framework, taking both the underlying logic and a set of proof methods as parameters, results in an environment for "declarative" natural deduction proofs that may be "executed" at the same time. Checking is achieved by the *Isar virtual machine* interpreter, which also provides an operational semantics of the Isar proof language.

Thus the Isar approach to readable formal proof documents is best characterized as being *interpretative*. It offers a higher conceptual level of formal proof that shall be considered as the new *primary* one. Any internal inferences taking place within the underlying deductive system are encapsulated in an abstract judgment of derivability. Any function mapping a proof goal to an appropriate proof rule may be incorporated as *proof method*. Thus arbitrary automated proof procedures may be integrated as opaque refinement steps.

This immediately raises the question of soundness, which is handled in Isar according to the *back-pressure* principle. The Isar/VM interpreter refers to actual inferences at the level below only *abstractly*, without breaching its integrity. Thus we inherit whatever notion of correctness is available in the underlying inference system (e.g. primitive proof terms). Basically, this is just the well-known "LCF approach" of correctness-by-construction applied at the level of the Isar/VM.

The issue of *presentation* of Isar documents can be kept rather trivial, because the proof language has been designed with readability already in mind. Some pretty printing and pruning of a few details should be sufficient for reasonable output. Nevertheless, Isar could be easily put into a broader context of more advanced presentation concepts, including natural language generation (e.g. [8]).

The rest of this paper is structured as follows. Section 2 presents some example proof documents written in the Isar language. Several aspects of Isar are discussed informally as we proceed. Section 3 reviews formal preliminaries required for the subsequent treatment of the Isar language. Section 4 introduces the Isar formal proof language syntax and operational semantics, proof methods, and extra-logical features.

2 Example Isar Proof Documents

Isar provides a generic framework for readable formal proofs that supports a broad range of both logics and proof tools. A typical instantiation for actual applications would use Higher-order Logic [6] together with a reasonable degree of automation [17, 18]. Yet the main objective of Isar is not to achieve shorter proofs with more automation, but better ones. The writer is enabled to express the interesting parts of the reasoning as explicit proof text, while leaving other parts to the machine. For the sake of the following examples, which are from pure first-order logic (both intuitionistic and classical), we refer to very simple proof methods only.

2.1 Basic Proofs

To get a first idea what natural deduction proofs may look like in Isar, we review three well-known propositions from intuitionistic logic: $I\colon A \to A$ and $K\colon A \to B \to A$ and $S\colon (A \to B \to C) \to (A \to B) \to A \to C$; recall that $\to$ is nested to the right.

```
theorem I: A → A
proof
  assume A
  show A .
qed
```

Unsurprisingly, the proof of I is rather trivial: we just assume A in order to show A again. The dot "." denotes an *immediate proof*, meaning that the current problem holds by assumption.

```
theorem K: A → B → A
proof
  assume A
  show B → A
  proof
    show A .
  qed
qed
```

Only slightly less trivial is K: we assume A in order to show $B \to A$, which holds because we can show A trivially. Note how proofs may be nested at any time, simply by stating a new problem (here via **show**). The subsequent proof (delimited by a **proof**/**qed** pair), is implicitly enclosed by a *logical block* that inherits the current context (assumptions etc.), but keeps any changes local. Block structure, which is a well-known principle from programming languages, is an important starting point to achieve structured proofs (e.g. [9]). Also note that there are implicit default proof methods invoked at the beginning (**proof**) and end (**qed**) of any subproof. The initial method associated with **proof** just picks standard introduction and elimination rules automatically according to the topmost symbol involved (here $\rightarrow$ introduction), the terminal method associated with **qed** solves all remaining goals by assumption. Proof methods may be also specified explicitly, as in "**proof** (*rule modus-ponens*)".

```
theorem S: $(A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$
proof
  assume $A \rightarrow B \rightarrow C$
  show $(A \rightarrow B) \rightarrow A \rightarrow C$
  proof
    assume $A \rightarrow B$
    show $A \rightarrow C$
    proof
      assume $A$
      show $C$
      proof (rule modus-ponens)
        show $B \rightarrow C$ by (rule modus-ponens)
        show $B$ by (rule modus-ponens)
      qed
    qed
  qed
qed
```

In order to prove S we first decompose the three topmost implications, represented by the **assume**/**show** pairs. Then we put things together again, by applying *modus ponens* to get C from $B \rightarrow C$ and B, which are themselves established by *modus ponens* (**by** abbreviates a single-step proof in terminal position). Note that the context of assumptions $A \rightarrow B \rightarrow C$, $A \rightarrow B$, A is taken into account implicitly where appropriate.

What have we achieved so far? Certainly, there are more compact ways to write down natural deduction proofs. In typed λ-calculus our examples would read $\lambda\, x\colon A.\ x$ and $\lambda\, x\colon A\ y\colon B.\ x$ and $\lambda\, x\colon A \rightarrow B \rightarrow C\ y\colon A \rightarrow B\ z\colon A.\ (x\, z)\, (y\, z)$.

The Isar text is much more verbose: apart from providing fancy keywords for arranging the proof, it explicitly says at every stage which statement is established next. Speaking in terms of λ-calculus, we have given types to actual subterms, rather than variables only. This sort of redundancy has already been observed in the ProveEasy teaching tool [5] as very important ingredient to improve readability of formal proofs. Yet one has to be cautious not to become too verbose, lest the structure of the reasoning be obscured again. Isar already leaves some inferences implicit, e.g. the way assumptions are applied. Moreover, the level of primitive inferences may be transcended by appealing to automated proof procedures, which are treated as opaque refinement steps (cf. §2.4).

2.2 Mixing Backward and Forward Reasoning

The previous examples have been strictly backward. While any proof may in principle be written this way, it may not be most natural. Forward style is often more adequate when working from intermediate facts.

Isar offers both backward and forward reasoning elements, as an example consider the following three proofs of $A \wedge B \rightarrow B \wedge A$.

```
lemma $A \wedge B \rightarrow B \wedge A$
proof
  assume $A \wedge B$
  show $B \wedge A$
  proof
    show $B$
      by ($rule\ conj_2$)
    show $A$
      by ($rule\ conj_1$)
  qed
qed
```

```
lemma $A \wedge B \rightarrow B \wedge A$
proof
  assume $A \wedge B$
  then
    show $B \wedge A$
  proof
    assume $A, B$
    show $??thesis$ ..
  qed
qed
```

```
lemma $A \wedge B \rightarrow B \wedge A$
proof
  assume $ab$: $A \wedge B$
  from $ab$
    have $a$: $A$ ..
  from $ab$
    have $b$: $B$ ..
  from $b, a$
    show $B \wedge A$ ..
qed
```

The first version is strictly backward, just as the examples of §2.1. We have to provide the projections $conj_{1,2}$ explicitly, because the corresponding goals do not provide enough syntactic structure to determine the next step. This may be seen as an indication that forward reasoning would be more appropriate.

Consequently, the second version proceeds by forward chaining from the assumption $A \wedge B$, as indicated by **then**. This corresponds to $\wedge$ elimination, i.e. we may assume the conjuncts in order to show again $B \wedge A$. Repeating the current goal is typical for elimination proofs, so Isar provides a way to refer to it symbolically as *??thesis*. The double dot ".." denotes a *trivial proof*, by a single standard rule. Alternatively, we could have written "**by** (*rule conj-intro*)".

Forward chaining may be done not only from the most recent fact, but from any one available in the current scope. This typically involves naming intermediate results (assumptions, or auxiliary results introduced via **have**) and referring to them explicitly via **from**. Thus the third proof above achieves an extreme case of forward-style reasoning, with only the outermost step being backward.

The key observation from these examples is that there is more to readable natural deduction than pure λ-calculus style reasoning. Isar's **then** language element can be understood as *reverse* application of λ-terms. Thus elimination proofs and other kinds of forward reasoning are supported as first-class concepts.

Leaving the writer the choice of proof direction is very important to achieve readable proofs, although yielding a nice balance between the extremes of purely forward and purely backward requires some degree of discernment. As a rule of thumb for good style, backward steps should be the big ones (decomposition, case analysis, induction etc.), while forward steps typically pick up assumptions or other facts to achieve the next result in a few small steps.

As a more realistic example for mixed backward and forward reasoning consider Peirce's law, which is a classical theorem so its proof is by contradiction. Backward-only proof would be rather nasty, due to the →-nesting.

```
theorem Peirce's-Law: ((A → B) → A) → A
proof                                                                  [¬A]
  assume ab-a: (A → B) → A                                               ⋮
  show A                                                                 A
  proof (rule contradiction)      −− use classical contradiction rule: ───
    assume not-a: ¬A                                                     A

    have ab: A → B
    proof
      assume a: A
      from not-a, a show B ..
    qed

    from ab-a, ab show A ..
  qed
qed
```

There are many more ways to arrange the reasoning. In the following variant we swap two sub-proofs of the contradiction. The result looks as if a *cut* had been performed. (The ⌈−⌋ parentheses are a version of **begin–end**.)

```
theorem Peirce's-Law: ((A → B) → A) → A
proof
  assume ab-a: (A → B) → A
  show A
  proof (rule contradiction)
  ⌈ assume ab: A → B              −− to be proved later (≈ cut)
    from ab-a, ab show A .. ⌋

    assume not-a: ¬A
    show A → B
    proof
      assume a: A
      from not-a, a show B ..
    qed
  qed
qed
```

Which of the two variants is actually more readable is a highly subjective question, of course. The most appropriate arrangement of reasoning steps also depends on what the writer wants to point out to the audience in some particular situation. Isar does not try to enforce any particular way of proceeding, but aims at offering a high degree of flexibility.

2.3 Intra-logical and Extra-logical Binding of Variables

Leaving propositional logic behind, we consider $(\exists x.\ P(f(x))) \to (\exists x.\ P(x))$. Informally, this holds since after assuming $\exists x.\ P(f(x))$, we may fix some a such that $P(f(a))$ holds, and use $f(a)$ as witness for x in $\exists x.\ P(x)$ (note that the two bound variables x are in separate scopes). So the proof is just a composition of $\to$ introduction, $\exists$ elimination, $\exists$ introduction. Writing down a natural deduction proof tree would result in a very compact and hard to understand representation of the reasoning involved, though. The Isar proof below tries to mimic our informal explanation, exhibiting many (redundant) details.

```
lemma $(\exists x.\ P(f(x))) \to (\exists x.\ P(x))$
proof
  assume $\exists x.\ P(f(x))$
  then show $\exists x.\ P(x)$
  proof (rule ex-elim)          -- use $\exists$ elimination rule:
    fix $a$
    assume $P(f(a))$ (is $P(??witness)$)
    show ??thesis by (rule ex-intro [with P ??witness])
  qed
qed
```

$$\frac{\exists x.\ A(x) \qquad \begin{array}{c}[A(x)]_x \\ \vdots \\ B\end{array}}{B}$$

After forward chaining from fact $\exists x.\ P(f(x))$, we have locally fixed an arbitrary a (via **fix**) and assumed that $P(f(a))$ holds. In order to stress the rôle of the constituents of this statement, we also say that $P(f(a))$ matches the pattern $P(??witness)$ (via **is**). Equipped with all these parts, the thesis is finally established using $\exists$ introduction instantiated with P and the *??witness* term.

Above example exhibits two different kinds of variable binding. First **fix** a, which introduces a local Skolem constant used to establish a quantified proposition as usual. Second (**is** $P(??witness)$), which defines a local abbreviation for some term by higher-order matching, namely $??witness \equiv f(a)$. The subsequent reasoning refers to a from *within* the logic, while abbreviations have a quite different logical status: being expanded before actual reasoning, the underlying logic engine will never see them. In a sense, this just provides an extra-logical illusion, yet a very powerful one.

Term abbreviations are also an important contribution to keep the Isar language lean and generic, avoiding separate language features for logic-specific proof idioms. Using appropriate proof methods together with abbreviations having telling names like *??lhs*, *??rhs*, *??case* already provides sufficient means for representing typical proofs by calculation, case analysis, induction etc. nicely. Also note that *??thesis* is just a special abbreviation that happens to be bound automatically — just consider any new goal goal implicitly decorated by (**is** *??thesis*). Note that "`thesis`" is a separate language element in Mizar [22].

2.4 Automated Proof Methods and Abstraction

The quantifier proof of §2.3 has been rather verbose, intentionally. We have chosen to provide proof rules explicitly, even given instantiations. As it happens, these rules and instantiations can be figured out by the basic mechanism of picking standard introduction and elimination rules that we have assumed as the standard initial proof method so far.

```
lemma $(\exists x.\ P(f(x))) \to (\exists x.\ P(x))$
proof
  assume $\exists x.\ P(f(x))$
  then show $\exists x.\ P(x)$
  proof
    fix $a$
    assume $P(f(a))$
    show $??thesis$ ..
qed
```

Much more powerful automated deduction tools have been developed over the last decades, of course. From the Isar perspective, any of these may be plugged into the generic language framework as particular proof methods. Thus we may achieve more abstract proofs beyond the level of primitive rules, by letting the system solve open branches of proofs automatically, provided the situation has become sufficiently "obvious". In the following version of our example we have collapsed the problem completely by a single application of method "*blast*", which shall refer to the generic tableau prover tactic integrated in Isabelle [18].

lemma $(\exists x.\ P(f(x))) \to (\exists x.\ P(x))$ **by** (*blast*)

Abstraction via automation gives the writer an additional dimension of choice in arranging proofs, yet a limited one, depending on the power of the automated tools available. Thus we achieve *accidental abstraction* only, in the sense that more succinct versions of the proof text still happen to work.

In practice, there will be often a conflict between the level of detail that the writer wishes to confer to his audience, and the automatic capabilities of the system. Isar also provides a simple mechanism for *explicit abstraction*. Subproofs started by **proof***/**by*** rather than **proof**/**by** are considered below the current *level of interest* for the intended audience. Thus excess detail may be easily pruned by the presentation component, e.g. printed as ellipsis ("…").

In a sense, **proof***/**by*** have the effect of turning concrete text into an ad-hoc proof method (which are always considered opaque in Isar). More general means to describe methods would include parameters and recursion. This is beyond the scope of Isar, though, which is an environment for writing actual proofs rather than proof methods. Isar is left computationally incomplete by full intention. High-level languages for proof methods are an issue in their own right [1].

3 Formal Preliminaries

3.1 Basic Mathematical Notations

Functions. We write function application as $f\,x$ and use λ-abstraction $\lambda\,x.\ f(x)$. Point-wise update of functions is written postfix, $f\,[x_1 := \ldots := x_n := y]$ denotes the function mapping $x_1, \ldots, x_n$ to y and any other x to $f(x)$. Sequential composition of functions f and g (from left to right) is written $f; g$ which is defined as $(f; g)(x) = g\ (f\ x)$. Any of these operations may be used both for total functions $(A \to B)$ and partial functions $(A \rightharpoonup B)$.

Records are like tuples with explicitly labeled fields. For any record $r\colon R$ with some field $a\colon A$ the following operations are assumed: selector $\mathit{get\text{-}a}\colon R \to A$, update $\mathit{set\text{-}a}\colon A \to (R \to R)$, and the functional $\mathit{map\text{-}a}\colon (A \to A) \to (R \to R)$ which is defined as $\mathit{map\text{-}a}\ f \equiv \lambda\, r.\ \mathit{set\text{-}a}\ (f\ (\mathit{get\text{-}a}\ r))$.

Lists. Let *list of* A be the set of lists over A. We write $[x_1, \ldots, x_n]$ for the list of elements $x_1, \ldots, x_n$. List operations include $x \circ \mathit{xs}$ (cons) and $\mathit{xs} \mathbin{@} \mathit{ys}$ (append).

3.2 Lambda-Calculus and Natural Deduction

Most of the following concepts are from λ-Prolog or Isabelle [16, Part I].

λ-Terms are formed as (typed) constants or variables (from set *var*), by application $t\ u$, or abstraction $\lambda\, x.\ t$. We also take α-, β-conversions for granted.

Higher-order Abstract Syntax. Simply-typed λ-terms provide a means to describe abstract syntax adequately. Syntactic entities are represented as types, and constructors as (higher-order) constants. Thus tree structure is achieved by (nested) application, variable binding by abstraction, and substitution by β-reduction.

Natural Deduction (Meta-logic). We consider a minimal $\Rightarrow$/$\forall$-fragment of intuitionistic logic. For the abstract syntax, fix type *prop* (meta-level propositions), and constants $\Rightarrow\colon \mathit{prop} \to \mathit{prop} \to \mathit{prop}$ (nested to the right), $\forall\colon (\alpha \to \mathit{prop}) \to \mathit{prop}$. We write $[\varphi_1, \ldots, \varphi_n] \Rightarrow \varphi$ for $\varphi_1 \Rightarrow \ldots \Rightarrow \varphi_n \Rightarrow \varphi$, and $\forall x.\ P\ x$ for $\forall(\lambda\, x.\ P\, x)$. Deduction is expressed as an inductive relation $\Gamma \vdash \varphi$, by the usual rules of assumption, and $\Rightarrow$/$\forall$ introduction and elimination. Note that the corresponding proof trees can be again seen as λ-terms, although at a different level, where propositions are types. The set "$\vdash$" is also called "*theorem*".

Encoding Object-logics. A broad range of natural deduction logics may now be encoded as follows. Fix types i of individuals, o of formulas, and a constant $D\colon o \to \mathit{prop}$ (for expressing derivability). Object-level natural deduction rules are represented as meta-level propositions, e.g. $\exists\colon (i \to o) \to o$ elimination as $D(\exists x.\ P\ x) \Rightarrow (\forall x.\ D(P\ x) \Rightarrow D(Q)) \Rightarrow D(Q)$. Let *form* be the set of propositions $D(A)$, for $A\colon o$. D is usually suppressed and left implicit. Object-level rules typically have the form of nested meta-level horn-clauses.

4 The Isar Proof Language

The Isar framework takes a meta-logical formulation (see §3.2) of the underlying logic as parameter. Thus we abstract over logical syntax, rules and automated proof procedures, which are represented as functions yielding rules (see §4.2).

4.1 Syntax

For the subsequent presentation of the Isar core syntax, *var* and *form* are from the underlying higher-order abstract syntax (see §3.2), *name* is any infinite set, while the *method* parameter refers to proof methods (see §4.2).

$$
\begin{aligned}
\textit{theory-stmt} &= \textbf{theorem}\ [\textit{name}\colon]\ \textit{form proof} \\
&\mid \textbf{lemma}\ [\textit{name}\colon]\ \textit{form proof} \\
&\mid \textbf{types}\ \dots \mid \textbf{consts}\ \dots \mid \textbf{defs}\ \dots \mid \dots \\
\textit{proof} &= \textbf{proof}\ [(\textit{method})]\ \textit{stmt}^*\ \textbf{qed}\ [(\textit{method})] \\
\textit{stmt} &= \textbf{begin}\ \textit{stmt}^*\ \textbf{end} \\
&\mid \textbf{note}\ \textit{name} = \textit{name}^+ \\
&\mid \textbf{fix}\ \textit{var}^+ \\
&\mid \textbf{assume}\ [\textit{name}\colon]\ \textit{form}^+ \\
&\mid \textbf{then}\ \textit{goal-stmt} \\
&\mid \textit{goal-stmt} \\
\textit{goal-stmt} &= \textbf{have}\ [\textit{name}\colon]\ \textit{form proof} \\
&\mid \textbf{show}\ [\textit{name}\colon]\ \textit{form proof}
\end{aligned}
$$

The actual Isar proof language (*proof*) is enclosed into a theory specification language (*theory-stmt*) that provides global statements **theorem** or **lemma**, entering into *proof* immediately, but also declarations and definitions of any kind. Note that advanced definitional mechanisms may also require proof. Enclosed by **proof**/**qed**, optionally with explicit initial or terminal method invocation, *proof* mainly consists of a list of local statements (*stmt*). This marginal syntactic rôle of *method* is in strong contrast to existing tactical proof languages.

Optional language elements above default to "it:" for result names, proof method specifications "(*single*)" for **proof**, "(*assumption*)" for **qed** (cf. §4.3).

A few well-formedness conditions of Isar texts are not yet covered by the above grammar. Considering **begin**–**end** and **show**/**have**–**qed** as block delimiters, we require any *name* reference to be well-defined in the current scope. Furthermore, the paths of variables introduced by **fix** may not contain duplicates, and **then** may only occur directly after **note**, **assume**, or **qed**.

Next the Isar core language is extended by a few derived elements. (Below *same* and *single* refer to standard proof methods introduced in §4.3.)

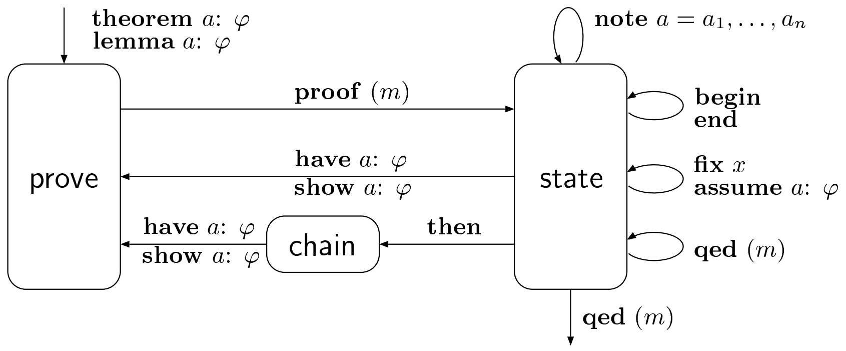
$$
\begin{aligned}
\textbf{from}\ a_1, \dots, a_n &\equiv \textbf{note}\ \mathsf{facts} = a_1, \dots, a_n\ \textbf{then} \\
\textbf{hence} &\equiv \textbf{then have} \\
\textbf{thus} &\equiv \textbf{then show} \\
\textbf{by}\ (m) &\equiv \textbf{proof}\ (m)\ \textbf{qed} && \text{("terminal proof")} \\
.. &\equiv \textbf{proof}\ (\textit{single})\ \textbf{qed} && \text{("trivial proof")} \\
. &\equiv \textbf{proof}\ (\textit{same})\ \textbf{qed} && \text{("immediate proof")}
\end{aligned}
$$

Basically, this is already the full syntax of the Isar language framework. Any logic-specific extensions will be by abbreviations or proof methods only.

4.2 Operational Semantics

The canonical operational semantics of Isar is given by direct interpretation within the *Isar virtual machine* (*Isar/VM*). Well-formed proof texts, which have tree structure if considered as abstract syntax entities, are translated into lists of Isar/VM instructions in a rather trivial way (the translation is particularly easy, because Isar lacks recursion): any syntactic construct, as indicated by some major keyword (**proof**, **begin**, **show**, etc.), simply becomes a separate instruction, which acts as transformer of the machine configuration.

Before going into further details, we consider the following abstract presentation of the Isar/VM. The machine configuration has three modes of operation, depicted as separate states prove, state, chain below. Transitions are marked by the corresponding Isar/VM instructions.



Any legal path of Isar/VM transitions constitutes an (interactive) proof, starting with **theorem/lemma** and ending eventually by a final **qed**. Intermediately, the two main modes alternate: prove (read "apply some method to refine the current problem") and state (read "build up a suitable environment to produce the next result"). Minor mode chain modifies the next goal accordingly.

More precisely, the Isar/VM configuration is a non-empty list of levels (for block structure), where each level consists of a record with the following fields:

$$\begin{aligned}
\mathit{mode} &: \mathsf{prove} \mid \mathsf{state} \mid \mathsf{chain} \\
\mathit{fixes} &: \mathit{list\ of\ var} \\
\mathit{asms} &: \mathit{list\ of\ form} \\
\mathit{results} &: \mathit{name} \rightharpoonup \mathit{list\ of\ theorem} \\
\mathit{problem} &: (\mathit{bool} \times \mathit{name}) \times ((\mathit{list\ of\ theorem}) \times \mathit{theorem}) \mid \mathsf{none}
\end{aligned}$$

Fields *fixes* and *asms* constitute the Skolem and assumption context. The *results* environment collects lists of intermediate theorems, including the special one "facts" holding the most recent result (used for forward chaining). An open *problem* consists of a flag (indicating if the finished result is to be used to refine an enclosing goal), the result's name, a list of facts for forward chaining, and the actual goal (for *theorem* see §3.2). Goals are represented as rules according to

Isabelle tradition [16, Part I]: $\Gamma \vdash [(\Phi_1 \Rightarrow \varphi_1), \ldots, (\Phi_n \Rightarrow \varphi_n)] \Rightarrow \chi$ means that in assumption context Γ, main goal χ is to be shown from n subgoals $\Phi_i \Rightarrow \varphi_i$.

An initial configuration, entered by **theorem** a: φ or **lemma** a: φ, has a single level with fields $\mathit{mode} = \mathsf{prove}$, $\mathit{fixes} = []$, $\mathit{asms} = []$, $\mathit{results} = \{(\mathsf{facts}, [])\}$, and $\mathit{problem} = ((\mathsf{false}, a), ([], \vdash \varphi \Rightarrow \varphi))$. The terminal configuration is $[]$. Intermediate transitions are by the (partial) function $\mathcal{T}[\![I]\!]$, mapping Isar/VM configurations depending on instruction I and the current mode value as follows. Basic record operations applied to a configuration refer to the topmost level.

```
$\mathit{mode} = \mathsf{prove}$ :
  $\mathcal{T}[\![\textbf{proof}\ (m)]\!] \equiv \mathit{refine\text{-}problem}\ m; \mathit{set\text{-}mode}\ \mathsf{state}$

$\mathit{mode} = \mathsf{state}$ :
  $\mathcal{T}[\![\textbf{note}\ a = a_1, \ldots, a_n]\!] \equiv$
      $\mathit{map\text{-}results}\ (\lambda\, r.\ r\ [\mathsf{facts} := a := r(a_1)\ @ \cdots @\ r(a_n)])$
  $\mathcal{T}[\![\textbf{begin}]\!] \equiv \mathit{reset\text{-}facts}; \mathit{open\text{-}block}; \mathit{set\text{-}problem}\ \mathsf{none}$
  $\mathcal{T}[\![\textbf{end}]\!] \equiv \mathit{close\text{-}block}$
  $\mathcal{T}[\![\textbf{fix}\ x]\!] \equiv \mathit{map\text{-}fixes}\ (\lambda\, xs.\ xs\ @\ [x]); \mathit{reset\text{-}facts}$
  $\mathcal{T}[\![\textbf{assume}\ a\text{: } \varphi_1, \ldots, \varphi_n]\!] \equiv$
      $\mathit{map\text{-}asms}\ (\lambda\, \Phi.\ \Phi\ @\ [\varphi_1, \ldots \varphi_n]);$
      $\mathit{map\text{-}results}\ (\lambda\, r.\ r\ [\mathsf{facts} := a := [\varphi_1 \vdash \varphi_1, \ldots, \varphi_n \vdash \varphi_n]])$
  $\mathcal{T}[\![\textbf{qed}\ (m)]\!] \equiv \mathit{refine\text{-}problem}\ m; \mathit{check\text{-}result}; \mathit{apply\text{-}result}; \mathit{close\text{-}block}$
  $\mathcal{T}[\![\textbf{then}]\!] \equiv \mathit{set\text{-}mode}\ \mathsf{chain}$
  $\mathcal{T}[\![\textbf{have}\ a\text{: } \varphi]\!] \equiv \mathit{setup\text{-}problem}\ (\mathsf{false}, a)\ (\mathsf{false}, \varphi)$
  $\mathcal{T}[\![\textbf{show}\ a\text{: } \varphi]\!] \equiv \mathit{setup\text{-}problem}\ (\mathsf{true}, a)\ (\mathsf{false}, \varphi)$

$\mathit{mode} = \mathsf{chain}$ :
  $\mathcal{T}[\![\textbf{have}\ a\text{: } \varphi]\!] \equiv \mathit{setup\text{-}problem}\ (\mathsf{false}, a)\ (\mathsf{true}, \varphi)$
  $\mathcal{T}[\![\textbf{show}\ a\text{: } \varphi]\!] \equiv \mathit{setup\text{-}problem}\ (\mathsf{true}, a)\ (\mathsf{true}, \varphi)$

$\mathit{open\text{-}block}\ (x \circ xs) \equiv x \circ x \circ xs$
$\mathit{close\text{-}block}\ (x \circ xs) \equiv xs$
$\mathit{reset\text{-}facts} \equiv \mathit{map\text{-}results}\ (\lambda\, r.\ r\ [\mathsf{facts} := []])$
$\mathit{setup\text{-}problem}\ \mathit{result\text{-}info}\ (\mathit{use\text{-}facts}, \varphi)\ c \equiv f\ c$   where
  $\Phi \equiv \mathit{get\text{-}asms}\ c$
  $\mathit{facts} \equiv \text{if } \mathit{use\text{-}facts} \text{ then } (\mathit{get\text{-}results}\ c\ \mathsf{facts}) \text{ else } []$
  $f \equiv \mathit{reset\text{-}facts}; \mathit{open\text{-}block}; \mathit{set\text{-}mode}\ \mathsf{prove};$
      $\mathit{set\text{-}problem}\ (\mathit{result\text{-}info}, (\mathit{facts}, \Phi \vdash (\Phi \Rightarrow \varphi) \Rightarrow \varphi))$
$\mathit{refine\text{-}problem}\ m \equiv$
  $\mathit{map\text{-}problem}\ (\lambda(x, (y, \mathit{goal})).\ (x, (y, \mathit{backchain}\ \mathit{goal}\ (m\ \mathit{facts}))))$
$\mathit{check\text{-}result}\ c \equiv c$   if $\mathit{problem}$ has no subgoals, else undefined
$\mathit{apply\text{-}result}\ (x \circ xs) \equiv x \circ (f\ xs)$   where
  $((\mathit{use\text{-}result}, a), (y, \mathit{res})) \equiv \mathit{get\text{-}problem}\ x$
  $\mathit{result} \equiv$
     $\mathit{generalise}\ (\mathit{get\text{-}fixes}\ x)\ (\mathit{discharge}\ (\mathit{get\text{-}asms}\ x - \mathit{get\text{-}asms}\ xs)\ \mathit{res})$
  $f \equiv \mathit{map\text{-}results}\ (\lambda\, r.\ r\ [\mathsf{facts} := a := [\mathit{result}]]);$
      $\text{if } \mathit{use\text{-}result} \text{ then } \mathit{refine\text{-}problem}\ (\lambda\, y.\ \mathit{result}) \text{ else } (\lambda\, x.\ x)$
```

Above operation *setup-problem* initializes a new problem according the current assumption context and forward chaining mode etc. The goal is set up to establish result φ from φ under the assumptions.

Operation *refine-problem* back-chains (using a form of meta-level *modus ponens*) with the method applied to the facts to be used for forward chaining.

Operation *apply-result* modifies the upper configuration, binding the result and refining the second topmost problem wrt. block structure (if *use-result* had been set). Note that *generalise* and *discharge* are basic meta-level rules.

We claim (a proof is beyond the scope of this paper) that the Isar/VM is correct and complete as follows. For any φ, there is a path of transitions from the initial to the terminal configuration *iff* $\vdash \varphi$ is a theorem derivable by natural deduction, assuming any rule in the image of the set of methods.

This result would guarantee that the Isar/VM does not fail unexpectedly, or produce unexpected theorems. Actual correctness in terms of formal derivability is achieved differently, though. Applying the well-known "LCF approach" of correctness-by-construction at the level of the Isar/VM implementation, results are always actual theorems, relative to the primitive inferences underlying proof methods and bookkeeping operations such as *refine-problem*.

4.3 Standard Proof Methods.

In order to turn the Isar/VM into an actually working theorem proving system, some standard proof methods have to be provided. We have already referred to a few basic methods like *same*, *single*, which are defined below.

We have modeled proof methods as functions producing appropriate rules (meta-level theorems), which will be used to refine goals by backchaining. Operationally, this corresponds to a function reducing goal $\Gamma \vdash \Phi \Rightarrow \chi$ to $\Gamma \vdash \Phi' \Rightarrow \chi$ (leaving the hypotheses and main goal unchanged). This coincides with tactic application, only that proof methods may depend on facts for forward chaining. For the subsequent presentation we prefer to describe methods according to this operational view.

Method "*same*" inserts the facts to any subgoal, which are left unchanged otherwise; "*rule a*" applies rule a by back-chaining, after forward-chaining from facts; "*single*" is similar to "*rule*", but determines a standard introduction or elimination rule from the topmost symbol of the goal or the first fact automatically; "*assumption*" solves subgoals by assumption.

These methods are sufficient to support primitive proofs as presented in §2. A more realistic environment would provide a few more advanced methods, in particular automated proof tools such as a generic tableau prover "*blast*" [18], a higher-order simplifier "*simp*" etc. A sufficiently powerful combination of such proof tools could be even made the default for **qed**. In contrast, the initial **proof** method should not be made too advanced by default, lest the subsequent proof text be obscured by the left-over state of its invocation. In DECLARE [21] automated *initial* proof methods are rejected altogether because of this.

4.4 Extra-logical Features

Isar is not a monolithic all-in-one language, but a hierarchy of concepts having different logical status. Apart from the core language considered so far, there are additional extra-logical features without semantics at the logical level.

Term Abbreviations. Any goal statement (**show** etc.) may be annotated with a list of term abbreviation patterns (**is** $pat_1 \ldots$ **is** pat_n). Alternatively, abbreviations may be bound by explicit **let** $pat \equiv term$ statements.

Levels of Interest decorate **proof**/**by** commands by a natural number or $\star$ (for infinity), indicating that the subsequent proof block becomes less interesting for the intended audience. The presentation component will use these hints to prune excess detail, collapsing it to ellipsis ("...") , for example.

Formal Comments of the form "$--$ *text*" may be associated with any Isar language element. The comment *text* may contain any text, which may again contain references to formal entities (terms, formulas, theorems etc.).

5 Conclusion and Related Work

We have proposed the generic *Intelligible semi-automated reasoning* (*Isar*) approach to readable formal proof documents. Its main aspect, the Isar formal proof language, supports both "declarative" proof texts and machine-checking, by direct "execution" within the Isar virtual machine (Isar/VM). While Isar does not require automation for basic operation, arbitrary automated deduction tools may be included in Isar proofs as appropriate. Automated tools are an important factor in scalability for realistic applications, of course.

Isar is most closely related to "declarative" theorem proving systems, notably Mizar [19, 22]. The Mizar project, started in the 1970s, has pioneered the idea of performing mathematical proof in a structured formal language, while hiding operational detail as much as possible. The gap towards the underlying calculus level is closed by a specific notion of *obvious inferences*. Internally, Mizar does not actually reduce its proof checking to basic inferences. Thus Mizar proofs are occasionally said to be "rigorous" only, rather than "formal".

Over the years, Mizar users have built up a large body of formalized mathematics. Despite this success, though, there are a few inherent shortcomings preventing further scalability for large applications. Mizar is not generic, but tightly built around its version of typed set theory and the *obvious inferences* prove checker. Its overall design has become rather baroque, such that even re-engineering the syntax has become a non-trivial effort recently. Also note that Mizar has a batch-mode proof checker only.

While drawing from the general experience of Mizar, Isar provides a fresh start that avoids these problems. The Isar concepts have been carefully designed with simplicity in mind, while preserving scalability. Isar is based on a lean hierarchic arrangement of basic concepts only. It is quite independent of the underlying logic and its automated tools, by employing a simple meta-logic framework. The Isar/VM interpretation process directly supports interactive development.

Two other systems have transferred Mizar ideas to tactical theorem proving. The "Mizar mode for HOL" [11] is a package that provides means to write tactic scripts in a way resembling Mizar proof text — it has not been developed beyond some initial experiments, though. DECLARE [20, 21] is a much more elaborate system experiment, which draws both from the Mizar and HOL tradition. It has been applied by its author to some substantial formalization of Java operational semantics [21]. DECLARE heavily depends on a its built-in automated procedure for proof checking, causing considerable run-time penalty compared to ordinary tactical proving. Furthermore, proofs cannot be freely arranged according to usual natural deduction practice.

Apart from "declarative" or "intelligible" theorem proving, there are several further approaches to provide human access to formal proof. Obviously, user interfaces for theorem provers (e.g. [3]) would be of great help in performing interactive proofs. Yet there is always a pending danger of overemphasizing advanced interaction mechanisms instead of adding high-level concepts to the underlying system. For example, *proof-by-pointing* offers the user a nice way to select subterms with the mouse. Such operations are rather hard to communicate later, without doing actual replay. In Isar one may select subterms more abstractly via term abbreviations, bound by higher-order matching.

Nevertheless, Isar may greatly benefit from some user interface support, like a *live document* editor that helps the writer to hierarchically step back and forth through the proof text during development. In fact, there is already a prototype available, based on the Edinburgh *ProofGeneral* interface. A more advanced system, should also provide high-level feedback and give suggestions of how to proceed — eventually resulting in *Computer-aided proof writing* (*CAPW*) as proposed in [21]. Furthermore, digestible information about automated proof methods, which are fully opaque in Isar so far, would be particularly useful. To this end, proof transformation and presentation techniques as employed e.g. in *Ω*Mega [2] appear to be appropriate. Ideally, the resulting system might be able to transform internal inferences of proof tools into the Isar proof language format. While this results in bottom-up generation of Isar proofs, another option would be top-down search of Isar documents, similar to the proof planning techniques that *Ω*Mega [2] strongly emphasizes, too. Shifting the focus even more beyond Isar towards actual high-level proof automation, we would arrive at something analogous to the combination of tactical theorem proving and proof planning undertaken in Clam/HOL [4].

We have characterized Isar as being "interpretative" — a higher level language is interpreted in terms of some lower level concepts. In contrast, transformational approaches such as [8] proceed in the opposite direction, abstracting primitive proof objects into a higher-level form, even natural-language.

Most of the Isar concepts presented in this paper have already been implemented within Isabelle. Isar will be part of the forthcoming Isabelle99 release. The system is already sufficiently complete to conduct proofs that are slightly more complex than the examples presented here. For example, a proof of Cantor's theorem that is much more comprehensive than the original tactic script discussed in [16, Part III]. Another example is correctness of a simple translator for arithmetic expressions to stack-machine instructions, formulated in Isabelle/HOL. The Isar proof document nicely spells out the interesting induction and case analysis parts, while leaving the rest to Isabelle's automated proof tools. More realistic applications of Isabelle/Isar are to be expected soon.

References

[1] K. Arkoudas. Deduction vis-a-vis computation: The need for a formal language for proof engineering. The MIT Express project, http://www.ai.mit.edu/projects/express/, June 1998.

[2] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. *Ω*Mega: Towards a mathematical assistant. In W. McCune, editor, *14th International Conference on Automated Deduction — CADE-14*, volume 1249 of *LNAI*. Springer, 1997.

[3] Y. Bertot and L. Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 11, 1996.

[4] R. Boulton, K. Slind, A. Bundy, and M. Gordon. An interface between CLAM and HOL. In J. Grundy and M. Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*. Springer, 1998.

[5] R. Burstall. Teaching people to write proofs: a tool. In *CafeOBJ Symposium, Numazu, Japan*, April 1998.

[6] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, pages 56–68, 1940.

[7] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, and C. Muñoz. *The Coq Proof Assistant User's Guide, version 6.1*. INRIA-Rocquencourt et CNRS-ENS Lyon, 1996.

[8] Y. Coscoy, G. Kahn, and L. Théry. Extracting text from proofs. In *Typed Lambda Calculus and Applications*, volume 902 of *LNCS*. Springer, 1995.

[9] 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.

[10] M. J. C. Gordon and T. F. Melham (editors). *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[11] J. Harrison. A Mizar mode for HOL. In J. Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'96*, volume 1125 of *LNCS*, pages 203–220. Springer, 1996.

[12] T. Nipkow. More Church-Rosser proofs (in Isabelle/HOL). In M. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction — CADE-13*, volume 1104 of *LNCS*, pages 733–747. Springer, 1996.

[13] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *LNCS*, pages 180–192. Springer, 1996.

[14] T. Nipkow and D. v. Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, New York, 1998.

[15] 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*. Springer, 1996.

[16] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[17] L. C. Paulson. Generic automatic proof tools. In R. Veroff, editor, *Automated Reasoning and its Applications*. MIT Press, 1997.

[18] L. C. Paulson. A generic tableau prover and its integration with Isabelle. In *CADE-15 Workshop on Integration of Deductive Systems*, 1998.

[19] P. Rudnicki. An overview of the MIZAR project. In *1992 Workshop on Types for Proofs and Programs*. Chalmers University of Technology, Bastad, 1992.

[20] D. Syme. DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, 1997.

[21] D. Syme. *Declarative Theorem Proving for Operational Semantics*. PhD thesis, University of Cambridge, 1998. Submitted.

[22] A. Trybulec. Some features of the Mizar language. Presented at a workshop in Turin, Italy, 1993.