

# Verifying and Invalidating Textbook Proofs using Scunak

Chad E. Brown

Universität des Saarlandes, Saarbrücken, Germany, cebrown@ags.uni-sb.de

**Abstract.** Many textbook proofs are essentially human-readable representations of natural deduction proofs. Terms in dependent type theory provide formally checkable representations of natural deduction proofs. We show how the new mathematical assistant system Scunak can be used to verify a textbook proof by translating the  $\text{\LaTeX}$  version into a proof term in a dependent type theory. We also show how Scunak can give interesting output upon failure.

## 1 Introduction

We use the new mathematical assistant system Scunak to analyze different versions of a textbook proof. We explain how Scunak can transform a  $\text{\LaTeX}$  version of the proof into a formally checkable proof term. Furthermore, in some cases, Scunak can signal an error and reject the proof. We use this case study to illustrate and discuss various issues related to the formalization of mathematics. For instance, in a textbook proof, how do we recognize if an assumption is “correct”? Also, how do we determine when an eigenvariable or hypothesis has been discharged? As we shall see, Scunak offers potential answers to such questions.

## 2 What is Scunak?

Scunak is like Automath [16] or Twelf [13] in the sense that the user creates a dependently typed signature of constants (corresponding to basic concepts and axioms) and definitions (corresponding to defined concepts and theorems with proofs). Scunak is like Coq [3] in that one can interactively create proof objects using commands corresponding to natural deduction in addition to applying previously proven facts. The type theory of Scunak is different from (and in most respects more restricted than) the type theories of Twelf, Automath and especially Coq. In particular, Scunak excludes (all forms of) polymorphism, and restricts to a fixed number of basic type families and second-order types. Also, though one is free to declare constants and definitions for whatever mathematical foundations one wants, Scunak does include built in support for signatures which include certain set theoretic concepts. So, Scunak is unlike Automath or Twelf in that set theory signatures have direct support in the system.

Scunak is also similar to MIZAR [15] in many respects. Scunak currently uses a form of untyped set theory (Mac Lane set theory with Universes). The

type theory allows the user to treat any class (relative to the set theory) like a type. MIZAR supports a different form of set theory (Tarski-Groethendieck), but this is not relevant here. MIZAR has a type structure, but prohibits empty types. This difference is best illustrated by the “class”  $\{x|x \in A\}$  where  $A$  is a set. In MIZAR, the type `Element of A` represents this class, unless  $A$  is empty, in which case `Element of A` consists of the empty set [4]. In Scunak, the corresponding class type `class (in A)` is empty if  $A$  is empty. (We usually simply write `A` for `class (in A)` and let the parser determine it is the class type of  $A$ .) There are pros and cons of allowing empty types which we do not discuss here. Another important distinction between Scunak and MIZAR is that Scunak currently has a library of about 300 theorems whereas MIZAR has a library of about 40,000 theorems [4]. It is important to point out that neither Scunak nor MIZAR is irrevocably tied to the particular set theory in which the mathematics is represented (as discussed in [15]).

Four goals were the most important in the design of Scunak:

1. **Naturality:** The mathematics should be represented in a natural way, similar (up to isomorphism) to what appears in mathematics texts.
2. **Formal Correctness:** The proofs should be formally checkable.
3. **Retrievability:** Retrieving theorems by content rather than by name must be possible.
4. **Automation:** Some reasonable degree of automated reasoning for proving theorems in the logic should be available.

We will illustrate the first three points in this paper by considering the formalization of a simple textbook proof from [2] (also considered in [1]). We will explain how Scunak can read the L<sup>A</sup>T<sub>E</sub>X representation of the proof, verify the proof, and construct a corresponding checkable proof term. During this process, Scunak must sometimes use facts in the signature in order to justify steps in the proof. We will also demonstrate how Scunak can read mutilated L<sup>A</sup>T<sub>E</sub>X versions of the proof and identify the error. We do not claim, however, Scunak is an “industrial strength” proofreader for mathematical proofs in L<sup>A</sup>T<sub>E</sub>X. The example is a vehicle for discussing various issues related to formalizing mathematics and for introducing Scunak as a tool for working with formalized mathematics.

### 3 A Simple Textbook Proof

We consider the first proof given in the introductory analysis book [2]. The proof (shown in Figure 1) is of the distributive law  $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$  (the first equation in part (d) of Theorem 1.1.4 in [2]). This proof was also considered in [1] (which was the source for the L<sup>A</sup>T<sub>E</sub>X version of the proof). The only difference between the L<sup>A</sup>T<sub>E</sub>X source of the proof in [1] and the L<sup>A</sup>T<sub>E</sub>X source discussed here is that the version in [1] used macros `\union` and `\inter` where the version checked by Scunak uses the standard macros `\cup` and `\cap`.

We begin by considering the structure of this proof. The first sentence states the intention to prove the distributive law. Scunak finds no pattern in this sentence with mathematical or proof content and so ignores the sentence. (Any text

In order to give a sample proof, we shall prove the first equation in (d). Let  $x$  be an element of  $A \cap (B \cup C)$ , then  $x \in A$  and  $x \in B \cup C$ . This means that  $x \in A$ , and either  $x \in B$  or  $x \in C$ . Hence we either have (i)  $x \in A$  and  $x \in B$ , or we have (ii)  $x \in A$  and  $x \in C$ . Therefore, either  $x \in A \cap B$  or  $x \in A \cap C$ , so  $x \in (A \cap B) \cup (A \cap C)$ . This shows that  $A \cap (B \cup C)$  is a subset of  $(A \cap B) \cup (A \cap C)$ .

Conversely, let  $y$  be an element of  $(A \cap B) \cup (A \cap C)$ . Then, either (iii)  $y \in A \cap B$ , or (iv)  $y \in A \cap C$ . It follows that  $y \in A$ , and either  $y \in B$  or  $y \in C$ . Therefore,  $y \in A$  and  $y \in B \cup C$  so that  $y \in A \cap (B \cup C)$ . Hence  $(A \cap B) \cup (A \cap C)$  is a subset of  $A \cap (B \cup C)$ .

In view of Definition 1.1.1, we conclude that the sets  $A \cap (B \cup C)$  and  $(A \cap B) \cup (A \cap C)$  are equal.

**Fig. 1.** Textbook Proof

which matches no rule in the proofreader’s context-free grammar is ignored.) The second sentence introduces  $x \in A \cap (B \cup C)$ . Why? A reader of the proof (say,  $R$ ) is aware of the goal of proving the sets  $A \cap (B \cup C)$  and  $(A \cap B) \cup (A \cap C)$  are equal.  $R$  should also know a common technique for proving such sets equal is to prove the two subset inclusions (relying on set extensionality, which is the content Definition 1.1.1 in [2]). Given this information,  $R$  can conclude that the author of the proof is introducing the eigenvariable  $x$  and hypothesis that  $x \in A \cap (B \cup C)$  holds in order to prove  $x \in (A \cap B) \cup (A \cap C)$  and thus conclude  $A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap C)$ . Thus  $R$  expects to read a proof of the goal  $x \in (A \cap B) \cup (A \cap C)$  and then read a proof of the other inclusion  $(A \cap B) \cup (A \cap C) \subseteq A \cap (B \cup C)$ . The mathematical content of the remainder of paragraph one, until the last sentence, outlines a series of facts which should follow from the previous facts and assumptions. The last sentence concludes  $A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap C)$ . Note that sentence does not explicitly say, “we discharge  $x$  and the hypothesis that  $x \in A \cap (B \cup C)$  in order to conclude...,” though this is clearly the logical structure.

The second paragraph introduces a new eigenvariable  $y$  and assumption  $y \in (A \cap B) \cup (A \cap C)$ .  $R$  should be expecting a proof of  $(A \cap B) \cup (A \cap C) \subseteq A \cap (B \cup C)$  and so such an assumption is warranted. Upon reading the assumption,  $R$  expects a proof that  $y \in A \cap (B \cup C)$ , which is exactly the content of the next three sentences. The last sentence implicitly discharges the  $y$  and corresponding assumption. The last sentence of the proof simply acknowledges that the proof is finished. We ignore the reference to “Definition 1.1.1.” Instead of ignoring this reference, a superior system would recognize “Definition 1.1.1” corresponds to particular facts or rules (such as the rule named `setextsub` discussed later) and ensure that such a rule is used in the justification.

The explanation above gives an intuitive idea of the behavior of Scunak upon reading the proof in Figure 1. Before giving a more detailed explanation, along

with examples of proofs with errors, we consider the Scunak type theory and set theory in which the content will be formally represented.

## 4 The Scunak Type Theory and Set Theory

The “logic” of Scunak is a modified version of the type theory LF as implemented in Twelf [13]. The meta-theory of LF is developed in [5]. The type theory of Scunak incorporates proof irrelevance for proof types. A more general framework for proof irrelevance is developed in [12] and [14]. A more thorough investigation of the Scunak type theory is planned for future work. Here we only sketch the type theory.

In the Scunak type theory, we restrict to three basic types:

- **obj** is the type of all (untyped) mathematical objects. Since we will be mainly interested in set theory, **obj** will be a synonym for **set**.
- **prop** is the type of mathematical propositions.
- **pf**  $P$  is a type whenever  $P$  is of type **prop** (in a context).

Such a restriction is not in the LF-tradition. The Scunak type theory is intended to model mathematical foundations instead of general logical and computational systems, so the restriction seems warranted.

One reason to insist on a single type of all mathematical objects is to avoid problems with polymorphism. However, sometimes one wants to consider a “typed” object. For this reason, we allow “class types.” **class**  $\phi$  is a type whenever  $\phi$  is a predicate (i.e., a function from **obj** to **prop**). Members of **class**  $\phi$  are pairs  $\langle x, p \rangle$  where  $x$  has type **obj** and  $p$  has type **pf**  $(\phi p)^\downarrow$ , where  $(\phi p)^\downarrow$  is a normal form of  $(\phi p)$ . Intuitively, **class**  $\phi$  consists of objects  $x$  along with a proof that  $x$  satisfies  $\phi$ . Since we do not want different proofs of the property to correspond to different elements of **class**  $\phi$ , we include proof irrelevance in the type theory. In particular, if  $M$  and  $N$  are of type **pf**  $P$ , then  $M$  and  $N$  are the *same* in **pf**  $P$ .

All other types are function types constructed using the the  $\Pi$  dependent type constructor.

We now describe the syntax in more detail. We use  $x, y, z, x^1, X, Y, \dots$  to denote variables and  $c, d, c^1, \dots$  to denote constants. We define terms and types as follows:

**Terms**  $M, N, P, Q, R, \phi, \dots := x|c|(\lambda x.M)|(M N)|\langle M, N \rangle|\pi_1(M)|\pi_2(M)$   
**Types**  $A, B, C, \dots := \mathbf{obj}|\mathbf{prop}|\mathbf{pf} P|\mathbf{class} \phi|\Pi x : A.B$

As usual we use  $(A \rightarrow B)$  to denote  $(\Pi x : A.B)$  when  $x$  does not occur free in  $B$ . Also, we identify terms and types up to  $\alpha$ -conversion. We assume all the usual notions of  $\lambda$ -calculus: substitution,  $\beta$ -reduction,  $\eta$ -reduction and the following pairing reductions:

$(\pi_1) : \pi_1(\langle M, N \rangle) \rightarrow_{\pi_1} M$        $(\pi_2) : \pi_2(\langle M, N \rangle) \rightarrow_{\pi_2} N$   
 $(\pi) : \langle \pi_1(M), \pi_2(M) \rangle \rightarrow_{\pi} M$

We say a term or type is normal if it contains no redexes. We write  $W^\downarrow$  for

the normal form of  $W$ , if a unique normal form of  $W$  exists. In practice we will only consider terms which are typable using simple types (in the Curry style [6]). This guarantees strong normalization and the Church-Rosser property whenever necessary, so that we can assume  $W^\downarrow$  exists uniquely.

In Scunak, terms and types are always given in  $\beta\pi_1\pi_2$ -normal form, so that the types of  $\lambda$ -abstractions and pairs can be inferred from the given intended type. We do not consider  $\eta$ -long (or  $\pi$ -long) forms. Instead, the rules for the typing judgments  $\eta$ - or  $\pi$ -expand on demand.

A signature  $\Sigma$  is a list of distinct constants associated with types. A type context  $\Gamma$  is a list of distinct variables associated with types. When  $\Gamma$  is  $z^1 : A^1, \dots, z^n : A^n$ , we may write  $\lambda\Gamma M$  for  $\lambda z^1 \dots \lambda z^n M$  and  $(M\Gamma)$  for  $(Mz^1 \dots z^n)$ .

We list the main typing judgments below, but omit the rules for space reasons.

- “ $\vdash \Sigma$  sig” Intuitively,  $\Sigma$  is a valid signature. The idea is to ensure  $\vdash_\Sigma A : Type$  before adding  $c : A$  to  $\Sigma$ .
- “ $\vdash_\Sigma \Gamma$  ctx” Intuitively,  $\Gamma$  is a valid context. The idea is to ensure  $\Gamma \vdash_\Sigma A : Type$  holds before adding  $x : A$  to  $\Gamma$ .
- “ $\Gamma \vdash_\Sigma M \sim N \uparrow A$ ” Intuitively,  $M$  can be checked to be  $A$ -related to  $N$ .
- “ $\Gamma \vdash_\Sigma M \sim N \downarrow A$ ” Intuitively, the type  $A$  can be extracted as a type in which  $M$  is  $A$ -related to  $N$ .
- “ $\Gamma \vdash_\Sigma A : Type$ ” In words,  $A$  is a valid type.

We say  $M$  has type  $A$  in context  $\Gamma$  if  $\Gamma \vdash M \sim M \uparrow A$  or  $\Gamma \vdash M \sim M \downarrow A$  holds. (The difference between extracting and checking types is a form of “bi-directional” algorithmic typing.) We usually omit the dependence on  $\Sigma$  in the judgments.

The Scunak type theory is further restricted to second-order. We consider `obj`, `prop`, `pf`  $P$  and `class`  $\phi$  to be first-order types. For all  $x : B \in \Gamma$ , we insist  $B$  is a second-order type, i.e., of the form  $\Pi x^1 : C^1 \dots \Pi x^n : C^n. C^{n+1}$  where each  $C^i$  is a first-order type. For all  $c : A \in \Sigma$ ,  $A$  must be a third-order type of the form  $\Pi x^1 : B^1 \dots \Pi x^n : B^n. C$  where each  $B^i$  is a second-order type and  $C$  is a first-order type.

The set theory is implemented in Scunak in the same spirit as the formal systems in [17]. In fact, the fact that many signatures in [17] are second-order was a motivation for considering that a second-order logical framework might be sufficient for representing mathematics.

The particular set theory is a form of Mac Lane set theory with universes which we will denote by **MU**. The kernel for this set theory is contained in a signature  $\Sigma_{\mathbf{MU}}$ . This signature currently contains 482 constants and abbreviations which includes the basic constructions and theorems leading to sets of functions. Of these 482 signature elements, 29 are constants entered by the user, 339 are abbreviations entered by the user, and 114 are constants for folding and unfolding definitions. These 114 constants are automatically generated by Scunak when the abbreviations are given. If one assumes  $\delta$ -reduction during type checking, the 114 generated constants are definable. (Due to proof irrelevance, one never needs to fold or unfold definitions returning a proof type. For this reason, constants for folding and unfolding such definitions are not generated.) We

only present enough details of the signature for the formalization of the textbook proof to make sense.

Two constants in  $\Sigma_{\text{MU}}$  are `eq` and `in`, both of type `obj  $\rightarrow$  obj  $\rightarrow$  prop`. As concrete syntax, Scunak allows infix notation `(A==B)` for `(eq A B)` and `(x::A)` for `(in A x)`. Note that `(in A x)` means  $x \in A$ . The reason for this choice is so that the  $\eta$ -short form `(in A)` corresponds to the class determined by the set `A`. In concrete syntax, we can use a set `A` as a type, though this is technically the class type `class (in A)`. Likewise, we can indicate this class type as `(in A)`. As noted earlier, if `A` is empty, this class type is empty.

There are abbreviations in  $\Sigma_{\text{MU}}$  corresponding to conjunction, disjunction, subset, binary union, and binary intersection. We use infix notations `&`, `|`, `<=`, `\cup`, and `\cap` as concrete syntax for these notions. When convenient, we may also display `<=`, `\cup` and `\cap` as  $\subseteq$ ,  $\cup$  and  $\cap$ , respectively.

There are also abbreviations corresponding to facts about these concepts. Consider the following two abbreviations (we only give the types):

- `setextsub :  $\Pi A : \text{obj} . \Pi B : \text{obj} . \Pi u : \text{pf } (A \subseteq B) . \Pi v : \text{pf } (B \subseteq A) . \text{pf } (A == B)$` . This constant can be used to form a proof that `A` equals `B` given two sets `A` and `B`, a proof of  $A \subseteq B$  and a proof of  $B \subseteq A$ .
- `subsetI1 :  $\Pi A : \text{obj} . \Pi B : \text{obj} . \Pi u : (\Pi x : \text{class } (\text{in } A) . \text{pf } (\text{in } B \ \pi_1(x))) . \text{pf } (A \subseteq B)$` . We can conclude  $A \subseteq B$  given two sets `A` and `B` and a function taking any `x` in the class type determined by `A` to a proof that the untyped part of `x` is an element of `B`.

In the concrete syntax when `x` is in a class type such as `class (in A)`, we can write `(x::B)` for `(in B  $\pi_1(x)$ )` since this can only be well-typed if we apply  $\pi_1$  to take `x` from being a member of a class type to being of type `obj`. Likewise, we need never explicitly write  $\pi_2(x)$  since the type constraints determine when `x` is being used as a proof type. Essentially, the  $\pi_1$  and  $\pi_2$  operators are reconstructed during parsing and the reconstructed term is type checked. Thus in any particular occurrence of a term `M` of class type, `M` may be used as a member of this class type, as an untyped object, or as a proof that the untyped object is in the class.

## 5 Verifying the Simple Textbook Proof

Following [8, 7], one can formalize a textbook proof via a three stage “formalization path.” First, translate from text to weak type theory (WTT). Second, translate this to a type theory with open terms (OTT). Third, instiate the meta-variables in open terms.

We follow an approach similar to [8, 7], though we do not work in stages. Instead, we extract certain commands from the text (`LATEX`) and these commands are executed by Scunak to affect the current proof state. At any point, the proof state corresponds to an open term with a stack of remaining tasks. In the end, the proof is completed if there are no remaining tasks (hence no remaining free variables). All this is performed when a user invokes the Scunak command `proofread` with a filename containing the `LATEX` source and a Scunak term corresponding to the formal version of the theorem which is to be proved.

Figure 2 shows the 14 Scunak commands extracted from the text in Figure 1 by the Scunak proofreader. These commands are in principle hidden from the user and are immediately executed upon being extracted from the text. The method of extracting these commands from the  $\text{\LaTeX}$  source is simple. A context-free grammar describes certain linguistic patterns which correspond to commands. For example, the rule

*PROOF1* :: *LET* \$*MATHVARIABLE*\$ be *A* *ATTRIBUTION*

is used to generate the first command in Figure 2. In this rule, *LET* may concretely be `Let` or `let`, and *A* may concretely be `a` or `an`. Note that we do not, for example, consider the parts-of-speech of the components of the sentences in the proof. We consider it an open question whether the mathematical content of texts can be better extracted using a large collection of specific rules such as the one above or using more sophisticated natural language processing techniques. The MathLang approach described in [9] appears to be somewhere in between the two extremes. In the end, experience with a large number of texts will determine the best way to extract the content automatically. Nothing significant with respect to language processing should be concluded from the simple proof in Figure 1.

1. `let x:(in (A \cap (B \cup C)))`.
2. `hence ((x::A) & (x::(B \cup C)))`.
3. `hence ((x::A) & ((x::B) | (x::C)))`.
4. `clearly (((x::A) & (x::B)) | ((x::A) & (x::C)))`.
5. `hence ((x::(A \cap B)) | (x::(A \cap C)))`.
6. `hence (x::((A \cap B) \cup (A \cap C)))`.
7. `hence ((A \cap (B \cup C)) <= ((A \cap B) \cup (A \cap C)))`.
8. `let y:(in ((A \cap B) \cup (A \cap C)))`.
9. `clearly ((y::(A \cap B)) | (y::(A \cap C)))`.
10. `hence ((y::A) & ((y::B) | (y::C)))`.
11. `hence ((y::A) & (y::(B \cup C)))`.
12. `hence (y::(A \cap (B \cup C)))`.
13. `hence (((A \cap B) \cup (A \cap C)) <= (A \cap (B \cup C)))`.
14. `clearly ((A \cap (B \cup C)) == ((A \cap B) \cup (A \cap C)))`

**Fig. 2.** Extracted Commands

The sequence of commands in Figure 2 can be fruitfully compared to the structured representation of the proof shown in Figure 3 of [1]. If one deletes the last line “4. Trivial” from Figure 3 of [1], then in both cases we have 14 lines corresponding to the same moves in the proof. However, [1] explicitly represents the scope of the assumptions  $x \in A \cap (B \cup C)$  and  $y \in (A \cap B) \cup (A \cap C)$ . This

information is not explicit in our representation. Note also that [1] is intended to be general with respect to formulas and terms, while we are using a concrete syntax for formulas and terms. The similarity between the two representations is not a coincidence. A close examination of the language in [1] (and its successors) inspired the parsing rules mentioned above.

One could use the parser to generate annotated text in the form of the language in [1], MathLang [9] or OMDOC [10]. However, this is a process independent of correctness of the proofs. In particular, the mutilated “proofs” described in Section 6 can be parsed and Scunak commands can be generated. Likewise, an annotated form of the mutilated “proofs” can be generated, even though the proofs may be technically incorrect. One can only judge correctness of proofs once one has a formal representation of the proposed proof and some formal notion of correctness. The Scunak type theory provides such a representation and such a notion of correctness. One could, of course, generate annotated text, then extract the Scunak commands for checking correctness from such annotated text. However, on such a simple proof there seems to be no motivation for using annotated text. One can simply regenerate the Scunak commands directly from the unannotated  $\text{\LaTeX}$  source.

The Scunak proofreader keeps a set of alternative states of the (open) formal proof. Each command in Figure 2 is evaluated with respect to each alternative proof state giving a set of alternative proof states as a result. If there are no resulting proof states after the command is executed, then a “verification failure” is reported, along with a message indicating where the failure occurred (in the  $\text{\LaTeX}$  source) and a general message indicating a reason for the failure.

An *open task* is  $\langle X, \Gamma, G \rangle$  where  $X$  is a variable,  $\Gamma$  is a type context, and  $G$  is a type containing no free variables outside of the context  $\Gamma$ . Intuitively, we wish to instantiate  $X$  with a closed term  $M$  such that  $(MT)^\downarrow$  has type  $G$  in context  $\Gamma$ . A *closed task* is  $\langle \Gamma, G \rangle$  where  $\Gamma$  is a type context, and  $G$  is a type containing no free variables outside of the context  $\Gamma$ . As we shall see, a closed task can be used for bookkeeping with respect to discharging eigenvariables and hypotheses. A *task* is either an open task or a closed task. A *proof state*  $S$  is an open proof term  $P$  with free variables  $X^1, \dots, X^n$  and a list of *tasks*, where each free variable corresponds to an open task.

There are certain global variables which control the behavior of the Scunak proofreader. These include two subsets of the signature  $\Sigma_{\text{MU}}$ . One subset  $\Sigma^u$  consists of signature elements corresponding to rules which can be used to try to justify steps in the proof. In analogy with first-order theorem proving, we say  $\Sigma^u$  is the *usable* set. Another subset  $\Sigma^e$  is a set of signature elements corresponding to rules which can be eagerly applied in the backwards direction. Only the elements in  $\Sigma^e$  can be used to apply a backwards step which introduces two new subgoals. The two sets  $\Sigma^u$  and  $\Sigma^e$  represent a model of the knowledge of the intended reader. Of course, to claim the proofreader is fully automatic, Scunak should choose the sets  $\Sigma^u$  and  $\Sigma^e$ . The proofreader is not fully automatic in this sense. We will consider different options for  $\Sigma^u$ , but we will explicitly restrict to the case where  $\Sigma^e$  contains only the rule `setextsub`. This rule is used

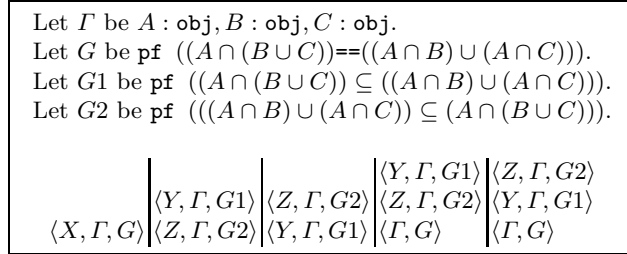
to prove an equation between two sets by proving the two inclusions, and must be applied eagerly before Scunak begins to read the proof. The choice of  $\Sigma^e$  is arguably where the user is supplying the most explicit information about how the proof should proceed.

We initialize the Scunak proofreader with a claim corresponding to the goal  $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$  in the context  $\Gamma$  where  $A$ ,  $B$ , and  $C$  are of type `obj` (equivalently, `set`). Let  $G$  be the type `pf`  $((A \cap (B \cup C)) = ((A \cap B) \cup (A \cap C)))$  of proofs of this proposition. Scunak creates a variable  $X$  of type  $\Pi A : \text{obj} . \Pi B : \text{obj} . \Pi C : \text{obj} . G$  and an open task  $\langle X, \Gamma, G \rangle$ .

One initial proof state consists of the open proof  $X$  and the single task  $\langle X, \Gamma, G \rangle$ . Using `setextsub`  $\in \Sigma^e$ , Scunak creates an open proof term

$$N := (\lambda A \lambda B \lambda C (\text{setextsub} \dots (Y ABC)(Z ABC)))$$

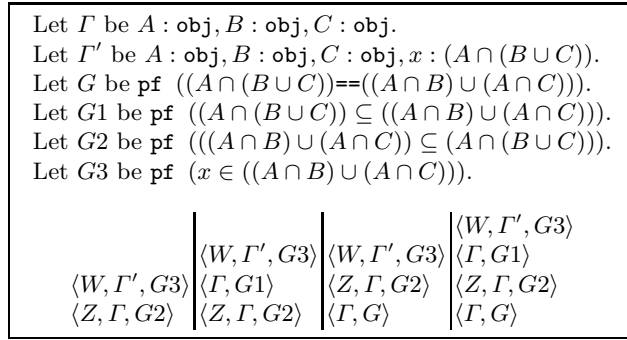
with new variables  $Y$  and  $Z$  corresponding to proofs of the two inclusions. Scunak creates four more proof states, all of which correspond to the open proof term  $N$ , but differ in the list of tasks. One alternative is to have two tasks corresponding to instantiating  $Y$  by proving  $(A \cap (B \cup C)) \subseteq ((A \cap B) \cup (A \cap C))$  and then instantiating  $Z$  by proving the other inclusion. A second alternative is to instantiate  $Z$  first by proving  $((A \cap B) \cup (A \cap C)) \subseteq (A \cap (B \cup C))$  and then instantiating  $Y$ . The third and fourth alternatives are the same as the first and second, except we include a third task: the closed task  $\langle \Gamma, G \rangle$ . As we shall see, this closed task must be included to check the final sentence of the proof of Figure 1. On the other hand, the proof in Figure 1 is valid if the final sentence is deleted, in which case the closed task should not be included. Hence we include alternatives with and without the closed task. In Figure 3, we show the five task stacks corresponding to the five initial proof states.



**Fig. 3.** Task Stacks of Initial Proof States

Scunak evaluates the first command in Figure 2 with the five alternative proof states described above (see Figure 3). The command corresponds to the text “Let  $x$  be an element of  $A \cap (B \cup C)$ ” in Figure 1. In order to evaluate this `let` command, the next task in the proof state should be an open task, and there must be some rule which introduces an eigenvariable of class type  $A \cap (B \cup C)$  and which

can be used to conclude the corresponding goal. For two alternatives, the current task is to instantiate  $Y$  by proving  $(A \cap (B \cup C)) \subseteq ((A \cap B) \cup (A \cap C))$ . Assuming the signature element `subsetI1` is in the usable set  $\Sigma^u$ , this element is found to have the appropriate type. For these alternatives, Scunak can imitate `subsetI1` for  $Y$  and create new alternatives. In each of the new alternatives, we create a version with a closed task corresponding to  $(A \cap (B \cup C)) \subseteq ((A \cap B) \cup (A \cap C))$  and a version without such a closed task. For the other three initial alternatives, there are no corresponding rules in  $\Sigma^u \subseteq \Sigma_{\mathbf{MU}}$ , so the alternatives have no successors after the command is executed. In particular, if the current goal is to prove  $((A \cap B) \cup (A \cap C)) \subseteq (A \cap (B \cup C))$ , then introducing  $x \in (A \cap (B \cup C))$  is clearly inappropriate. Hence we see that Scunak can determine when making an assumption (or introducing an eigenvariable) is a “correct” step. After the first command, it becomes clear that we are first proving the inclusion  $(A \cap (B \cup C)) \subseteq ((A \cap B) \cup (A \cap C))$  by proving  $x \in ((A \cap B) \cup (A \cap C))$ .



**Fig. 4.** Task Stacks of Proof States after First Command

After the first command is evaluated, there are four alternatives. All four have a proof term corresponding to using `setextsub` and `subsetI1`, but differ in the inclusion or exclusion of closed tasks. In all four cases, the next task is an open task corresponding to proving  $x \in ((A \cap B) \cup (A \cap C))$  in a context with  $x : (A \cap (B \cup C))$  (i.e.,  $x$  in the class type determined by  $(A \cap (B \cup C))$ ). We display the task stacks for these alternative proof states in Figure 4, using  $W$  for the new variable.

The second command in Figure 2 is a `hence` command. A command `hence`  $P$  can be interpreted in two different ways, depending on whether the next task is open or closed. If the next task is closed and the closed task has type **pf**  $P$ , then a new proof state is created by removing this closed task from the list. (Commands 7, 13, and 14 in Figure 2 provide examples of this behavior.) On the other hand, if the next task is an open task  $\langle X', \Gamma', G' \rangle$ , then Scunak attempts to justify the fact by providing a closed term of type **pf**  $P$ . In order to justify this fact, some propositional reasoning is applied (removing some conjunctions

and disjunctions) and then searching the usable set  $\Sigma^u$  for signature elements which can fill remaining gaps (without introducing new gaps). Suppose the gap is filled with a term  $Q$ . In the new proof state,  $X'$  is instantiated with  $\lambda\Gamma(X''\Gamma Q)$  where  $X''$  is a new variable and the task  $\langle X', \Gamma', G' \rangle$  is replaced by the task  $\langle X'', \Gamma'', G' \rangle$ , where  $\Gamma''$  is  $\Gamma', w : \text{pf } P$ . The idea is that we still have the goal of showing  $G'$ , but now we have (a proof of)  $P$  in the context of this goal.

In our particular example, the fact that  $x \in A$  and  $x \in (B \cup C)$  holds follows by propositional reasoning (conjunction introduction) along with two rules `binintersectEL` and `binintersectER` which take  $x \in (A \cap (B \cup C))$  to  $x \in A$  and  $x \in (B \cup C)$ . Note that these rules must be in the usable set in order for the proofreader to justify this step.

Commands 3-6 in Figure 2 are all used to conclude a certain fact, as with the second command. (The only difference between `clearly` and `hence` is that `hence` indicates that particular attention should be paid to the last fact added to the context.)

After the sixth command is executed, there are several alternatives. For some of these alternatives, the next task is a closed task corresponding to  $(A \cap (B \cup C)) \subseteq ((A \cap B) \cup (A \cap C))$ . Command 7 in Figure 2 creates proof states by removing this closed task. Note that this has the effect of discharging the  $x$  and the hypothesis that  $x$  is in  $(A \cap (B \cup C))$ . The other alternatives disappear after the seventh command is executed.

Commands 8-12 correspond to proving the other inclusion and follow a similar pattern as commands 1-6. As with command 7, command 13 simply notes the end of this part of the proof (technically by deleting a closed task). Finally, command 14 notes the end of the proof by deleting the last closed task.

Since the closed tasks can be included or excluded, one can delete some of the sentences in the textbook proof (those corresponding to commands 7, 13, and 14) and Scunak can still verify the proof. Also, one sentence in the first paragraph (corresponding to command 4) can be deleted since this is only involves propositional reasoning. The shortened proof is shown in Figure 5. One could say that the proof in Figure 5 is at an appropriate level of granularity for Scunak. (This shortened version can be contrasted to the longer “patched” version in Figure 6 of [1].)

<p>Let <math>x</math> be an element of <math>A \cap (B \cup C)</math>, then <math>x \in A</math> and <math>x \in B \cup C</math>. This means that <math>x \in A</math>, and either <math>x \in B</math> or <math>x \in C</math>. Therefore, either <math>x \in A \cap B</math> or <math>x \in A \cap C</math>, so <math>x \in (A \cap B) \cup (A \cap C)</math>.</p> <p>Conversely, let <math>y</math> be an element of <math>(A \cap B) \cup (A \cap C)</math>. Then, either (iii) <math>y \in A \cap B</math>, or (iv) <math>y \in A \cap C</math>. It follows that <math>y \in A</math>, and either <math>y \in B</math> or <math>y \in C</math>. Therefore, <math>y \in A</math> and <math>y \in B \cup C</math> so that <math>y \in A \cap (B \cup C)</math>.</p>
--

**Fig. 5.** Shortened Textbook Proof

In order for Scunak to verify the proof, the usable set  $\Sigma^u$  must contain at least the rule for introducing subset and rules for introducing and eliminating binary unions and intersections. We ran the Scunak proofreader with three possible settings of  $\Sigma^u$ . First, with  $\Sigma_0^u$  equal to eight elements corresponding to the rules we require about subset, binary union and binary intersection. Second, with  $\Sigma_1^u$  containing all 79 rules which mention subset, binary union or binary intersection. Third, with  $\Sigma_2^u$  containing all 435 rules in the **MU**-kernel. In the table below, we show the time (in seconds) taken to verify the two proofs using each usable set. From these results, it is clear that the usable set makes a significant difference.

	$\Sigma_0^u$	$\Sigma_1^u$	$\Sigma_2^u$
Fig 1 Proof	2	62	457
Fig 5 Proof	3	20	251

Upon completing the verification, Scunak outputs the proof term. This proof term can be easily type-checked without any need for search.

## 6 Finding Bugs in Textbook Proofs

For the purpose of building a mathematical library, the most interesting proofs are correct proofs which give new facts for the formal library. On the other hand, when proofreading, one is usually searching for mistakes. We briefly consider eight erroneous “proofs” obtained by mutilating the proof in Figure 1. Instead of repeating the entire proof, we indicate the change introducing the error and describe the output of the Scunak proofreader. Computing a “reason” for the error can be problematic because the “reason” may be different in different alternative proof states. Scunak simply collects reasons and outputs the one which occurs the most often.

1. Change the first sentence so “then  $x \in A$  and  $x \in B \cup C$ ” reads “then  $x \in B$  and  $x \in B \cup C$ .” Scunak indicates that “ $x \in B$  and  $x \in B \cup C$ ” may not follow.
2. Change “Therefore, either  $x \in A \cap B$  or  $x \in A \cap C$ ,” to be “Therefore, either  $x \in A \cap B$  or  $x \in A \cup C$ ,” in fifth sentence. In this case, Scunak indicates that the statement *following* the mutilated statement cannot be verified. In particular, Scunak can verify the mutilated statement “ $x \in A \cap B$  or  $x \in A \cup C$ ,” but cannot afterwards verify “so  $x \in (A \cap B) \cup (A \cap C)$ .”
3. Change the conclusion of the first paragraph to be “This shows that  $A \cap (B \cup C)$  is a subset of  $(A \cup C) \cap (B \cup C)$ .” Scunak indicates that we have not shown this conclusion.
4. In the second paragraph, change “(iii)  $y \in A \cap B$ ” to be “(iii)  $x \in A \cap B$ .” Scunak signals that “Then, either (iii)  $x \in A \cap B$ , or (iv)  $y \in A \cap C$ ” is not obvious. (Signalling that  $x$  is out of context would be preferable.)
5. Change the conclusion of the last sentence to read “ $A \cap (B \cup C)$  and  $(A \cap B) \cup (C \cap A)$  are equal” (commuting  $A$  and  $C$ ). Scunak signals that the last sentence does not follow.

6. In the second sentence, change “Let  $x$  be an element of  $A \cap (B \cup C)$ ” to be “Let  $x$  be an element of  $A \cup (B \cap C)$ ” to corrupt the assumption about  $x$ . Scunak indicates that the type given for  $x$  does not seem to be correct.
7. Change the first sentence of the second paragraph to read “Conversely, let  $y$  be an element of  $(A \cap C) \cup (B \cap C)$ .” Scunak indicates that the type given for  $y$  does not seem to be correct.
8. Remove the sentence “This means that  $x \in A$ , and either  $x \in B$  or  $x \in C$ .” The resulting proof is, in a sense, missing a step. Scunak indicates that the next sentence is not obvious.

We can also consider incomplete proofs. In such a case, the Scunak proofreader will not contain a complete proof upon termination, and will simply output a message indicating that the proof is not complete. Of course, an incomplete proof may also contain an error, in which case Scunak will signal the error before signalling that the proof is incomplete. One need not have a complete proof before invoking the proofreader.

## 7 Conclusion

Scunak can be used to proofread a simple mathematical proof written in L<sup>A</sup>T<sub>E</sub>X. For Scunak to proofread the proof the basic rules must be part of the usable subset of the full signature. More work on improving unification and indexing is required to handle the case when the usable set is large.

Because the proof of distributivity is so simple, it can act as a first touchstone for proposed approaches to checking proofs in texts. Any reasonable approach for verifying mathematical text should be able not only to verify this proof, but also find the errors in the mutilated versions of the proof. Then different approaches can be compared on a common, easy-to-understand problem.

The example also makes it clear when approaches are *not* intended to verify mathematical text. That is, any system which can read the proof in its mutilated forms without signalling an error is not intended to verify the proof content of mathematical text. Such a distinction is obviously important for understanding the intention of different systems.

*Acknowledgements.* Thanks to Magdalena Wolska and Alberto González for helpful discussions about parsing natural language.

## References

1. Serge Autexier and Armin Fiedler. Textbook proofs meet formal logic - the problem of underspecification and granularity. In Michael Kohlhase, editor, *Proceedings of MKM'05*, volume 3863 of *LNAI*, IUB Bremen, Germany, January 2006. Springer.
2. R.G. Bartle and D.R. Sherbert. *Introduction to Real Analysis*. John Wiley & Sons, New York, 1982.
3. Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

4. Adam Grabowski and Christoph Schwarzweiler. Translating mathematical vernacular into knowledge repositories. In Kohlhase [11], pages 49–64.
5. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
6. J. Roger Hindley. *Basic Simple Type Theory*. Cambridge University Press, 1997.
7. Gueorgui I. Jojgov. Translating a fragment of weak type theory into type theory with open terms. In Kohlhase [11], pages 389–403.
8. Gueorgui I. Jojgov and Rob Nederpelt. A path to faithful formalizations of mathematics. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *MKM*, volume 3119 of *Lecture Notes in Computer Science*, pages 145–159. Springer, 2004.
9. Fairouz Kamareddine, Manuel Maarek, and J. B. Wells. Toward an object-oriented structure for mathematical text. In Kohlhase [11], pages 217–233.
10. Michael Kohlhase. Omdoc: Towards an internet standard for the administration, distribution, and teaching of mathematical knowledge. In John A. Campbell and Eugenio Roanes-Lozano, editors, *Artificial Intelligence and Symbolic Computation: International Conference AISC 2000*, volume 1930 of *Lecture Notes in Artificial Intelligence*, pages 32–52. Springer-Verlag, 2001.
11. Michael Kohlhase, editor. *Mathematical Knowledge Management, 4th International Conference, MKM 2005, Bremen, Germany, July 15-17, 2005, Revised Selected Papers*, volume 3863 of *Lecture Notes in Computer Science*. Springer, 2006.
12. Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In Joseph Halpern, editor, *Proceedings of the Sixteenth Annual IEEE Symp. on Logic in Computer Science, LICS 2001*, pages 221–. IEEE Computer Society Press, June 2001.
13. Frank Pfenning and Carsten Schürmann. System Description: Twelf—A Meta-Logical Framework for Deductive Systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206, Trento, Italy, 1999. Springer-Verlag.
14. Jason Reed. Proof irrelevance and strict definitions in a logical framework. Technical Report 02-153, School of Computer Science, Carnegie Mellon University, 2002.
15. Piotr Rudnicki and Andrzej Trybulec. On the integrity of a repository of formalized mathematics. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *MKM*, volume 2594 of *Lecture Notes in Computer Science*, pages 162–174. Springer, 2003.
16. Freek Wiedijk. A new implementation of Automath. *J. Autom. Reasoning*, 29(3-4):365–387, 2002.
17. Freek Wiedijk. Is ZF a hack? Comparing the complexity of some (formalist interpretations of) foundational systems for mathematics. *Journal of Applied Logic*, 4, 2006. to appear.