A SYNTHESIS OF THE PROCEDURAL AND DECLARATIVE STYLES OF INTERACTIVE THEOREM PROVING

FREEK WIEDIJK

Institute for Computing and Information Sciences, Radboud University Nijmegen, Heyendaalseweg 135, 6525 AJ Nijmegen, The Netherlands
e-mail address: freek@cs.ru.nl

ABSTRACT. We propose a synthesis of the two proof styles of interactive theorem proving: the procedural style (where proofs are scripts of commands, like in Coq) and the declarative style (where proofs are texts in a controlled natural language, like in Isabelle/Isar). Our approach combines the advantages of the declarative style – the possibility to write formal proofs like normal mathematical text – and the procedural style – strong automation and help with shaping the proofs, including determining the statements of intermediate steps.

Our approach is new, and differs significantly from the ways in which the procedural and declarative proof styles have been combined before in the Isabelle, Ssreflect and Matita systems. Our approach is generic and can be implemented on top of any procedural interactive theorem prover, regardless of its architecture and logical foundations.

To show the viability of our proposed approach, we fully implemented it as a proof interface called miz3, on top of the HOL Light interactive theorem prover. The declarative language that this interface uses is a slight variant of the language of the Mizar system, and can be used for any interactive theorem prover regardless of its logical foundations. The miz3 interface allows easy access to the full set of tactics and formal libraries of HOL Light, and as such has 'industrial strength'.

Our approach gives a way to automatically convert any procedural proof to a declarative counterpart, where the converted proof is similar in size to the original. As all declarative systems have essentially the same proof language, this gives a straightforward way to port proofs between interactive theorem provers.

1. INTRODUCTION

1.1. Proof styles of interactive theorem proving.

Interactive theorem provers, also known as proof assistants, are computer programs for the development and verification of mathematical texts in a formal language. These systems make it *certain*[1] that the verified mathematics contains no errors at all. The activity of coding mathematics in the formal

[1]There is only one serious possibility for mathematics verified with the best interactive theorem provers to still have problems [43, 22]. The definitions and statements might not mean what the person who wrote them thinks they mean. Although it then still is *certain* that the mathematics contains no errors at all, it is the 'wrong' mathematics.

language of an interactive theorem prover is called *formalizing*, and the set of resulting
input files for such a system is called a *formalization*. Highly non-trivial proofs have been
formalized, both in mathematics [17, 29] and in computer science [16, 33, 34].

In interactive theorem proving one can consider the proofs on two different levels.
There is the *user level proof*, the proof in the formalization files on the level of which the
user interacts with the system. And there is the *proof object*, the proof in the formal system
underlying the system. Generally the second is an order of a magnitude larger than the
former. In systems like Coq and HOL [19], the user level proof consists of a list of tactics
to be executed. The proof objects in Coq are lambda terms, while in HOL they consist of
traces of function calls into the LCF style kernel of the system. In a system like Mizar [20]
the user level proof consists of the proof steps in the input language of the system. The
Mizar implementation does not keep track of proof objects, but the proofs on the proof
object level would be the formal deductions in first order predicate logic.

Interactive theorem provers can use three different proof styles (the terminology origi-
nates in [26]):

(1) *The procedural style.* In these systems the user inputs a proof as a sequence of *tactic*
    invocations, which are commands that transform proof obligations called *goals*. A
    tactic reduces a goal to zero or more new subgoals. When all goals have been solved
    this way, the proof is finished. Note that although most procedural systems support
    both forward and backward proof, the user interaction in those system primarily
    consists of reasoning backwards from a goal. Systems that use the procedural style
    are the various HOL systems like HOL4 [19], HOL Light [23, 27] and ProofPower
    [35], the original version of Isabelle [38], Coq [13], Matita [3], PVS [39], the B method
    [1] and Metamath [37].

    Some procedural systems offer the option to print their proof objects in the form
    of natural language text. An example of such a system is Matita (see the discussion
    in Section 1.3 below).

(2) *The declarative style.* In these systems the user inputs proofs in a stylized natural
    deduction language (de Bruijn called such a language a *mathematical vernacular*
    [15]). The output of the system then consists of messages that point out where the
    proof text still has errors. There are two subclasses of this proof style:

    (a) *The natural language declarative style.* Here the proof language is a formal
        version of mathematical natural language, also called a *controlled* natural lan-
        guage. Trivial reasoning between the steps in a proof is provided by the system
        through (light weight) automation. Systems that use this proof style are Mizar
        [20] and Isabelle with its 'structured' proof language Isar [49, 50].
        Actually, the Mizar and Isar languages, as well as the language described in
        this paper, are not very much like natural language. The ForTheL language
        for the SAD system [40] is much better in this respect.

    (b) *The proof object declarative style.* Here the proof input language is a syntactic
        rendering of the proof object, with the structure of a natural deduction proof.
        Systems that use this style are Twelf [42], Agda [2] and Epigram [36].
        In some of these systems, the user does not need to type the whole input text
        by themselves, but can also give *commands* in the interface that generate part
        of the proof text. These commands will *not* be part of the proof files that are
        the final formalization. Examples of such systems are Agda and Epigram.

(3) *The guided automated style.* In these systems the input is a sequence of lemma statements, which the system then tries to prove by itself. These systems generally produce long natural language texts for each lemma describing how it was proved. Often for some of the lemmas parameters need to be given that direct the system how to perform the proof. Also the lemma statements need to be chosen well for the system to be able to do the proof, as the gaps between the statements should not be too large. For this reason these systems still are *interactive* theorem provers. Systems that use this style are ACL2 [32] and Theorema [10].

One can consider the guided automated style to be either an extreme version of the procedural or an extreme version of the declarative style. In a guided automated theorem prover, one runs one supertactic per lemma. Or, in a guided automated theorem prover one writes a theory as a series of lemma statements, where the system checks that each statement follows from the previous ones.

It is interesting to compare where the 'proof commands to the system' and where the 'natural language proofs' are in these various proof styles:

	commands	*natural language*
procedural	input	absent or output
declarative, natural language	absent	input
declarative, proof object	absent or interface	absent
guided automated	input	output
the system from this paper	interface *and* input	input *and* output

There are some systems that are outside the simplicity of this table, like Isabelle with its combination of natural language declarative and procedural proof styles, like Matita with its declarative proof checker, and like the 'skeletonizer' of Mizar. These systems will be discussed in Section 1.3 below.

The proof style that we propose in this paper is an integration of the procedural and natural language declarative styles. The advantage of the declarative style is that it is closer to normal mathematical practice (one just writes proofs) with more readable proof scripts, and that it gives full control over the exact statements in the proof. Also declarative proofs tend to be easier to maintain and less dependent on the specific system than procedural ones. The advantage of the procedural style is that one does not need to write all intermediate statements: these are generated automatically. Also, procedural systems tend to have much stronger automation, with often many different *decision procedures* that without human help can perform proofs in specific domains.

The goal of this paper is to propose a proof style that combines the best of these two worlds.

1.2. **Relating the procedural and declarative proof styles.** Looked at superficially, the procedural and declarative proof styles seem very different. Certainly the proof scripts for those two styles *look* completely different. However, when working with these systems, both styles turn out to have a very similar work flow.

When working on a declarative proof, most of the time when one is not finished the only errors left are that the system did not succeed in proving some of the steps in the proof from the earlier steps. In Mizar these steps are called *unjustified* steps, and have error numbers *1 and *4. Now these unjustified steps correspond exactly to the subgoals

that one looks at when a procedural proof is not finished! In other words, an unfinished proof of a Mizar lemma in which there still are – say – seven unjustified steps left, is very similar to a Coq or HOL proof in which there are still seven subgoals left. This is the first observation that is the basis of our approach.

A proof in a declarative system consists of 'steps', most of which contain a statement. If one does the analogous proof in a procedural system, one goes through many subgoals that *also* consist of many statements. Now it turns out that those two sets of statements in practice are very much the same! In other words, if for a procedural proof we collect all the statements in the goals (both the assumptions and the statements to be proved), then those statements can in a natural way be organized as a declarative proof. This is the second observation at the basis of our approach.

Note that in these two observations there is no reference to proof objects. This means that our integration of the procedural and declarative proof styles does not have anything to do with proofs on the proof object level. It also means that our proposed approach is independent of the foundations or architecture of the system. What we propose will apply to *any* system in which the user performs proofs by executing tactics on subgoals containing statements.

Our proposal then is to have a proof interface in which a user is working on a declarative proof. In this proof the unjustified statements *are* considered the subgoals of the prover. At any of these steps/subgoals one can execute any tactic of the system, and if this is successful the statements of the new subgoals that the tactic produces will be merged into the proof text, making the declarative proof 'grow' [31]. However, one also can freely manually edit and then recheck the declarative text. The text does not need to 'remember' how it has been grown.

For an example of how all this works out in a concrete session, see Section 2 below.

1.3. **Related Work.** From the introduction in Section 1.1 it will be clear that the proof style that we propose is a combination of aspects of many different proof systems. However, there are some systems that are quite close to what we propose. For each of them we will discuss now how they differ from our work:

Isabelle/Isar: In Isabelle one can encapsulate procedural proof fragments consisting of tactic applications in a declarative Isar proof text [49, 50]. However, the user needs to manually type the declarative text (it is not generated like in our approach), and the procedural proofs do not make sense without running them on the system (unlike in our approach, where the tactics do not *solve* the goal but connect statements together).

Ssreflect: The usage of Georges Gonthier's Ssreflect language for Coq [18] is similar to a common way of using Isar. It is used declaratively for the high level structure of the proof while at the 'leaves' of the proof the user switches to the procedural proof style. However, the declarative part of Ssreflect is much less developed than Isar. Also, although Ssreflect is clearly intended to be also used declaratively, it barely fits category (2a) in the classification above.

HELM/MoWGLI/Matita: The HELM, MoWGLI and Matita systems [3, 4, 5] have as one of their goals to render type theoretical proof objects as natural language. In Matita, these rendered proof objects also can be read back in, and checked for correctness like in a declarative proof system [45].

An important difference with our approach is that one cannot go back from declarative editing to procedural proving. Once a declarative proof text has been modified, if the procedural proof from which it has been generated also gets modified, both modifications cannot be integrated. In other words, once one has worked declaratively, working procedurally is no longer possible.

Another difference is that the declarative proof text is generated from the proof object, which is generally more fine grained than the user level proof on the level of the tactic invocations, and is therefore more verbose and less understandable.

The proof rendering from the Lemme project: When proofs are being rendered as natural language, the source of the rendering is generally the proof object. An important exception is a system by Frédérique Guilhot, Hanane Naciri and Loïc Pottier. Unfortunately, this work seems not to have been published, all that exists is a set of slides for a talk about it [21].

A difference with our approach is that the generated text cannot be modified by the user anymore. The rendering in this system is just output, and is not parsed back again.

NuPRL: The NuPRL system [12] has a way to display formal proofs in which groups of tactics are interleaved with fragments of goals. Between the groups of tactics, the parts of the goal that have changed are shown (see for an example the NuPRL chapter in [54]). This is quite similar to what happens in our approach.

But again, this rendering is just output, and is not parsed back again.

Mizar's skeletonizer: In natural language declarative systems like Isar and Mizar, the proof text has to be written by the user. A slight exception to this is the 'skeletonizer' of the emacs interface to Mizar by Josef Urban [47]. In this interface, a proof skeleton is automatically generated from the statement to be proved.

This is similar to what happens when we 'grow' a proof by executing a tactic. However, in our case the growing is *generic*: for each tactic there is a corresponding way to insert part of the proof. In Mizar there is only one such way.

There already are various declarative proof languages which have been grafted on top of a procedural system. Currently Isabelle/Isar is the only one that knows widespread use. Others are:

Several 'Mizar modes' for HOL Light: There are two by John Harrison [24, 28] and two earlier ones by the author ([52] and an unpublished one included in the HOL Light distribution [23]). The `3` in the system name `miz3` refers to the fact that this is the third Mizar mode for HOL Light that we developed.

C-zar: A declarative proof language for Coq by Pierre Corbineau [14].

PhoX: There exists an experimental declarative version of the PhoX theorem prover by Christophe Raffalli [44].

These systems are all quite similar. The main improvement of the work described in this paper over these other systems is that it adds a Mizar-style interaction model, and that it integrates execution of tactics with generation of proof text.

1.4. **Contribution.** This paper is a continuation of the work in [31, 52]. It contains three contributions:

- We describe a declarative proof interface for the HOL Light theorem prover that is much more developed and far more ergonomic than earlier attempts at this. This software can be downloaded at:

$$\text{http://www.cs.ru.nl/}^\sim\text{freek/miz3/miz3.tar.gz}$$

- We describe a new proof style for interactive theorem provers that is a synthesis between the procedural and declarative proof styles.
- We describe a method for automatically converting *any* existing procedural proof to a declarative equivalent. This gives an approach for conserving libraries of formal proofs and semi-automatically porting them between systems. For details see Section 5 below.

1.5. **Outline.** The structure of the paper is as follows. In Section 2 we describe through an example how the proof interface that we developed works. In Section 3 we describe the declarative proof language of this interface. In Section 4 we give some details of the implementation of this interface. In Section 5 we describe how our approach makes it possible to automatically convert existing proofs to our language. In Section 6 we describe our experiences with using our interface on a non-trivial example. Finally in Section 7 we conclude with some observations and planned future work.

2. The miz3 proof interface to HOL Light

We developed a prototype of the interface style proposed in this paper as a layer called miz3 on top of the HOL Light system [23, 27]. It consists of about 2,000 lines of OCaml code (in comparison, the basic HOL Light system is approximately 30,000 lines), and its development took three man months.

We will explain the miz3 interface with a simple example. For this we will use the traditional inductive proof of

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

In HOL Light this is written as:

$$\texttt{!n. nsum (1..n) (\textbackslash i. i) = (n * (n + 1)) DIV 2}$$

In this formula, the exclamation mark ! is ASCII for the universal quantifier $\forall$, the backslash \ is ASCII for the λ of function abstraction, and the function nsum is a higher order version of the summation operator $\sum$.

Of course the example is very trivial. We do not want to give the impression that our approach only works well for simple examples like this. We just chose this example to be able to fit a reasonable representation of a proof session for it in the paper. We worked on much larger proofs with miz3, and it performs well on those too. For a description of such a larger example see Section 6 below.

To prove the equality of the example, we will need one lemma, which is the recursive characterization of nsum. The statement of this lemma, which we will use for rewriting expressions involving nsum, is:

```
# NSUM_CLAUSES_NUMSEG;;
val it : thm =
  |- (!m. nsum (m..0) f = (if m = 0 then f 0 else 0)) /\
     (!m n.
          nsum (m..SUC n) f =
          (if m <= SUC n then nsum (m..n) f + f (SUC n) else nsum (m..n) f))
```

This is the first example of a command in a HOL Light session. We indicate user input by putting boxes around it, to differentiate it from the output from the system that is outside those boxes.

We will now show how the proof of this statement is developed, both in the traditional procedural style of the HOL Light system, as well in the synthesis between the procedural and declarative proof styles that we propose in the paper.

The example using the procedural proof style of HOL Light. Traditionally, one develops the proof of a lemma in HOL Light in an interactive session. However, the exact commands from that session are not what is put in the formalization file. We now first show the session, and then the proof as it is written in the file.

The session for this lemma consists of six commands, with after each command output from the system:

```
# g '!n. nsum(1..n) (\i. i) = (n*(n + 1)) DIV 2';;
val it : goalstack = 1 subgoal (1 total)

'!n. nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2'

# e INDUCT_TAC;;
val it : goalstack = 2 subgoals (2 total)

  0 ['nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2']

'nsum (1..SUC n) (\i. i) = (SUC n * (SUC n + 1)) DIV 2'

'nsum (1..0) (\i. i) = (0 * (0 + 1)) DIV 2'

# e (ASM_REWRITE_TAC[NSUM_CLAUSES_NUMSEG]);;
val it : goalstack = 1 subgoal (2 total)

'(if 1 = 0 then 0 else 0) = (0 * (0 + 1)) DIV 2'

# e ARITH_TAC;;
val it : goalstack = 1 subgoal (1 total)

  0 ['nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2']

'nsum (1..SUC n) (\i. i) = (SUC n * (SUC n + 1)) DIV 2'

# e (ASM_REWRITE_TAC[NSUM_CLAUSES_NUMSEG]);;
val it : goalstack = 1 subgoal (1 total)
```

```
  0 [`nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2`]

`(if 1 <= SUC n then (n * (n + 1)) DIV 2 + SUC n else (n * (n + 1)) DIV 2) =
 (SUC n * (SUC n + 1)) DIV 2`
```

```
# e ARITH_TAC;;
```
```
val it : goalstack = No subgoals
```

The session starts with a `g` command that sets the goal to be proved, and then executes five tactics using the `e` command. Each time after a tactic, the system presents the subgoals that the tactic produced, where the assumptions from which the subgoal has to be proved are numbered (from 0), and the statement to be proved is unnumbered. If there are multiple subgoals produced (as is the case with `INDUCT_TAC`), the first goal to be worked on is printed last.

The proof as it appears in the formalization file uses a more compact representation of these commands. It consists of just three lines, containing the name of the lemma, the statement, and the sequence of tactics separated by `THEN`s:

```
let ARITHMETIC_SUM = prove
 (`!n. nsum(1..n) (\i. i) = (n*(n + 1)) DIV 2`,
  INDUCT_TAC THEN ASM_REWRITE_TAC[NSUM_CLAUSES_NUMSEG] THEN ARITH_TAC);;
```

This is the customary shape of lemmas in a HOL Light formalization.

The example using the `miz3` proof style. We will now show how the same proof is developed using the `miz3` interface. This is a synthesis of the procedural and declarative proof styles, but more specifically it is a close synthesis of the HOL Light and Mizar proof styles. For example, like in Mizar the system will modify the file being worked on by putting error messages inside that file. Also, the syntax of the `miz3` proofs (explained in detail in Section 3 below) is a direct hybrid of the syntax of Mizar and HOL Light: the proof steps are written using Mizar syntax, but the formulas and types in those proof use HOL Light syntax.

There are three ways to process a `miz3` proof. First, one can just give the proof text as a command to the OCaml interpreter running HOL Light. The parser has been modified to recognize the following convention:

quotation style	is parsed as
` ‘ … ‘ `	a term
` ‘ : … ‘ `	a type
` ‘ ; … ‘ `	a proof

Second, one can put the proof (without the backquotes and semicolon) in a file with suffix `.mz3` and check that using the checking program `miz3`. Third, one can use the interface from the `vi` editor. The third style is the one we will explain in the rest of this section. For all three interaction styles to work, a HOL Light session with the `miz3` code loaded has to be running, as a 'server'. One runs a HOL Light session in the `miz3` source directory, and in it executes the command:

```
# #use "miz3.ml";;
```
⟨*many lines of output*⟩

Once a HOL Light session is running with `miz3` loaded, one can have it check `miz3` proofs from a `vi` editor session that is running in a different window (there does not even have to be a file). When typing two keystrokes, *control-*`C` and then *return*, the part of the file where the cursor is, in between two empty lines, is checked. We will denote these keystrokes by:

control-`C` *return*

If there are errors in the checked part of the file, appropriate error messages are inserted. For example, an unfinished proof with errors messages inserted might look like:

```
let ARITHMETIC_SUM = thm ';
  !n. nsum(1..n) (\i. i) = (n*(n + 1)) DIV 2
  proof
    nsum(1..0) (\i. i) = 0 [1];
::                                #2
:: 2: inference time-out
    now let n be num;
      assume nsum(1..n) (\i. i) = (n*(n + 1)) DIV 2;
      thus nsum(1..SUC n) (\i. i) = (SUC n*(SUC n + 1)) DIV 2;
::                                                      #2
    end;
  qed by INDUCT_TAC,1;
::                    #1
:: 1: inference error
';;
```

The first error is caused by the lemma `NSUM_CLAUSES_NUMSEG` not being referenced, the second error has the same reason but is also caused by the proof automation of the system not being strong enough, and the third error is caused by the base case being wrong: it should not be `... = 0` but `... = (0*(0 + 1)) DIV 2`.

If one manually writes a proof for the lemma, checking for errors all the time and fixing them until no errors remain (this is the style of working with the Mizar system), one ends up with a proof like:

```
let ARITHMETIC_SUM = thm ';
  !n. nsum(1..n) (\i. i) = (n*(n + 1)) DIV 2
  proof
    nsum(1..0) (\i. i) = 0 by NSUM_CLAUSES_NUMSEG;
      .= (0*(0 + 1)) DIV 2 [1];
    now let n be num;
      assume nsum(1..n) (\i. i) = (n*(n + 1)) DIV 2 [2];
      1 <= SUC n;
      nsum(1..SUC n) (\i. i) = (n*(n + 1)) DIV 2 + SUC n
        by NSUM_CLAUSES_NUMSEG,2;
      thus .= ((SUC n)*(SUC n + 1)) DIV 2;
    end;
  qed by INDUCT_TAC,1';;
```

This proof is thirteen lines instead of the three that we got with the traditional proof style. I.e., this proof is quite a bit longer, but not unreasonably so. We experimented quite a bit by comparing procedural proofs to their declarative counterparts, and our impression is that declarative proofs are generally about twice as long as corresponding procedural ones. See in this respect also the statistics in Section 6 on page 22 below.

This proof was written using `miz3` in a purely declarative style. We now show how one can use `miz3` in a purely procedural style, exactly mimicking the traditional HOL Light session. (We have the problem of how to present an interactive editing session on paper. We will do this by presenting various stages of the edit buffer, interspersed with comments. This will mean a lot of duplicated text, but hopefully it will make the process clear.)

The standard starting point for a 'procedural' `miz3` session is:

```
let  = thm ';

  proof
  qed by #;
';;
```

In the place of the two empty spaces one puts the name and statement of the lemma to be proved:

```
let  ARITHMETIC_SUM  = thm ';
   !n. nsum(1..n) (\i. i) = (n*(n + 1)) DIV 2
  proof
  qed by #;
';;
```

When checking this, there will be *no* error messages added, as the `#` mark means that this line is a subgoal to be proved. The `qed` step means that the statement from the lemma has been proved, and therefore the subgoal in this case consists of exactly that statement.

Next, one types a tactic after the `#` sign and has the system process the file:

```
let ARITHMETIC_SUM = thm ';
  !n. nsum(1..n) (\i. i) = (n*(n + 1)) DIV 2
  proof
  qed by # INDUCT_TAC control-C return ;
';;
```

The tactic will be executed, and the system will 'merge' the two subgoals that are generated (see the traditional session on page 7) into the proof, using the method from [31]. Also, the insertion point of the editor will be put directly after the first `#` as indicated by the vertical bar, i.e., the editor will 'jump' to the first subgoal that is now left:

```
let ARITHMETIC_SUM = thm ';
  !n. nsum(1..n) (\i. i) = (n*(n + 1)) DIV 2
  proof
    nsum (1..0) (\i. i) = (0 * (0 + 1)) DIV 2 [1] by #;
    !n. nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2
        ==> nsum (1..SUC n) (\i. i) = (SUC n * (SUC n + 1)) DIV 2 [2]
    proof
      let n be num;
```

```
        assume nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2;
      qed by #;
    qed by INDUCT_TAC from 1,2;
`;;
```

There is quite some text already, but most of it is generated and not typed by the user. Next one enters the second tactic, and has the system process it:

```
let ARITHMETIC_SUM = thm `;
  !n. nsum(1..n) (\i. i) = (n*(n + 1)) DIV 2
  proof
    nsum (1..0) (\i. i) = (0 * (0 + 1)) DIV 2 [1] by #REWRITE_TAC,NSUM_CLAUSES_NU
MSEG control-C return;
    !n. nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2
        ==> nsum (1..SUC n) (\i. i) = (SUC n * (SUC n + 1)) DIV 2 [2]
    proof
      let n be num;
      assume nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2;
    qed by #;
  qed by INDUCT_TAC from 1,2;
`;;
```

This leads to a new subgoal, again exactly matching the one from the traditional HOL Light session:

```
let ARITHMETIC_SUM = thm `;
  !n. nsum(1..n) (\i. i) = (n*(n + 1)) DIV 2
  proof
    (if 1 = 0 then 0 else 0) = (0 * (0 + 1)) DIV 2 [1] by #;
    nsum (1..0) (\i. i) = (0 * (0 + 1)) DIV 2 [2]
      by REWRITE_TAC,NSUM_CLAUSES_NUMSEG from 1;
    !n. nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2
        ==> nsum (1..SUC n) (\i. i) = (SUC n * (SUC n + 1)) DIV 2 [3]
    proof
      let n be num;
      assume nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2;
    qed by #;
  qed by INDUCT_TAC from 2,3;
`;;
```

Note that the system automatically wrapped the line with the long tactic.

If one continues like this by putting in three more tactics, one gets the declarative counterpart of the procedural proof:

```
let ARITHMETIC_SUM = thm `;
  !n. nsum(1..n) (\i. i) = (n*(n + 1)) DIV 2
  proof
    (if 1 = 0 then 0 else 0) = (0 * (0 + 1)) DIV 2 [1] by ARITH_TAC;
    nsum (1..0) (\i. i) = (0 * (0 + 1)) DIV 2 [2]
      by REWRITE_TAC,NSUM_CLAUSES_NUMSEG from 1;
    !n. nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2
        ==> nsum (1..SUC n) (\i. i) = (SUC n * (SUC n + 1)) DIV 2 [3]
```

```
    proof
      let n be num;
      assume nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2;
      (if 1 <= SUC n
       then nsum (1..n) (\i. i) + SUC n
       else nsum (1..n) (\i. i)) =
      (SUC n * (SUC n + 1)) DIV 2 [4] by ARITH_TAC;
    qed by REWRITE_TAC,NSUM_CLAUSES_NUMSEG from 4;
  qed by INDUCT_TAC from 2,3;
‘;;
```

This is not a proof a human would have written, but it *exactly* matches the traditional
HOL Light session. Furthermore, the characters that one needs to type to get it are almost
exactly the same as in that session.

Now the point of this paper is that one does *not* need to choose between these two ways
of working. One can work on a proof in a combination, both freely editing it and rechecking
things while going, but also by executing tactics at various places and using the new proof
steps that these tactics produce.

Note that miz3 just replaces the *proof* part of the HOL Light mathematical language.
The rest of HOL Light's features like definitions and the implementation of proof automation
are unchanged. Therefore, for convenience the $\boxed{control\text{-}\mathsf{C}\ return}$ command will interpret text
being processed as straight OCaml code if it does not have the shape

```
    ... thm ‘;
      ...
    ‘;;
```

where the ‘; has to be at the end of the first line of the block. This means that there is
no need to enter text directly in the HOL Light session, one can fully work from the vi
interface. However, working in the HOL Light session itself is also possible.

3. The miz3 proof language

The three most common proof systems for first order predicate logic are natural deduction,
sequent calculus and Hilbert-style logic. Of these three, natural deduction corresponds most
closely with everyday mathematical reasoning. The two main proof systems for natural
deduction are Jaśkowski/Fitch- and Gentzen-style deduction [41]. Of these, the first is the
easiest to use for actual proofs. An example of a proof in a Jaśkowski/Fitch-style natural
deduction proof system (where we use the syntactic conventions of [30]) is:

$$
\begin{array}{rll}
1 & (\exists x\,\neg P(x)) \vee \neg(\exists x\,\neg P(x)) & \text{LEM} \\
2 & \exists x\,\neg P(x) & \text{assumption} \\
3 & x \quad \neg P(x) & \text{assumption} \\
4 & \quad P(x) & \text{assumption} \\
5 & \quad \bot & \neg E\ 3,4 \\
6 & \quad \forall y\,P(y) & \bot E\ 5 \\
7 & \quad P(x) \Rightarrow \forall y\,P(y) & \Rightarrow I\ 4\text{–}6 \\
8 & \quad \exists x(P(x) \Rightarrow \forall y\,P(y)) & \exists I\ 7 \\
9 & \exists x(P(x) \Rightarrow \forall y\,P(y)) & \exists E\ 2,3\text{–}8 \\[1em]
10 & \neg(\exists x\,\neg P(x)) & \text{assumption} \\
11 & \quad P(a) & \text{assumption} \\
12 & y \quad P(y) \vee \neg P(y) & \text{LEM} \\
13 & \quad P(y) & \text{assumption} \\
14 & \quad \neg P(y) & \text{assumption} \\
15 & \quad \exists x\,\neg P(x) & \exists I\ 14 \\
16 & \quad \bot & \neg E\ 10,15 \\
17 & \quad P(y) & \bot E\ 16 \\
18 & \quad P(y) & \vee E\ 12,13,14\text{–}17 \\
19 & \quad \forall y\,P(y) & \forall I\ 12\text{–}18 \\
20 & \quad P(a) \Rightarrow \forall y\,P(y) & \Rightarrow I\ 11\text{–}19 \\
21 & \quad \exists x(P(x) \Rightarrow \forall y\,P(y)) & \exists I\ 20 \\
22 & \exists x(P(x) \Rightarrow \forall y\,P(y)) & \vee E\ 1,2\text{–}9,10\text{–}21
\end{array}
$$

This is a proof of Smullyan's 'drinker's principle' [46].

Most declarative systems have essentially the same proof language (although the *explanation* of the meaning of the elements of such a language can differ significantly between systems [51].) In other words, declarative interactive theorem provers have proof languages that are much more similar to each other than the languages of procedural systems are. It turns out that the languages of declarative systems are all close to Jaśkowski/Fitch-style natural deduction.

The `miz3` proof language is in particular almost identical to the language of the Mizar system [20]. The natural deduction example that we just gave becomes in the syntax of the `miz3` language:

```
let DRINKER = thm ';
  let P be A->bool;
  thus ?x. P x ==> !y. P y [22]
  proof
    (?x. ~P x) \/ ~(?x. ~P x) [1];
    cases by 1;
    suppose ?x. ~P x [2];
      consider x such that ~P x [3] by 2;
      take x;
      assume P x [4];
      F [5] by 3,4;
      thus !y. P y [6] by 5;
    end;
```

```
    suppose ~(?x. ~P x) [10];
      consider a being A such that T;
      take a;
      assume P a [11];
      let y be A;
      P y \/ ~P y [12];
      cases by 12;
      suppose P y [13];
        thus P y by 13;
      end;
      suppose ~P y [14];
        ?x. ~P x [15] by 14;
        F [16] by 10,15;
        thus P y [17] by 16;
      end;
    end;
  end`;;
```

(The extra 'consider a being A such that T;' step is needed because in miz3 variables
have to be introduced before they can be used. The Jaśkowski/Fitch-style natural deduction
proof can use the free variable a without introducing it, but in miz3 this is not allowed. The
step should be read as 'consider an object a of type A such that true holds'. It is accepted by
the system because in HOL all types are non-empty. This non-emptiness is indeed essential
for the drinker's principle to be provable.)

This example shows the correspondence between miz3 proof steps and natural deduction
rules:

deduction rule	proof step
assumption	thus
$\wedge I$	thus
$\Rightarrow I$	assume
$\neg I$	assume
$\forall I$	let
$\exists I$	take
$\vee E$	cases/suppose
$\exists E$	consider

The other natural deduction rules, like $\wedge E$, $\Rightarrow E$, $\neg E$, $\forall E$ and $\vee I$, are all subsumed by the
inference checker 'by'. Actually, the $\wedge I$ and $\exists I$ rules are subsumed by by as well. The thus
and take steps are more of a backward step than a direct counterpart to the $\wedge I$ and $\exists I$
rules.

We now present the meaning of the miz3 language. It is almost exactly the same as
that of the Mizar language [20]. At every step in a proof there is a designated statement
called the thesis. This is the statement that is being proved (the *goal* of a procedural
prover). Subproofs have their own local thesis. Now steps can add extra variables and
statements to the proof context, but can also change the thesis. If that happens the step
is called a *skeleton* step. Here is a table that summarizes the main miz3 proof steps:

proof step	*statement to be justified*	*new proof variables*	*referable statement*	*old* `thesis`	*new* `thesis`	*skeleton step?*
`φ by ...;`	ϕ		ϕ	ψ	ψ	$-$
`let` x_1 `...` x_n `be` τ`;`		$x_1 \ldots x_n$		$\forall x_1 \ldots \forall x_n \, \psi$	ψ	$+$
`assume φ;`			ϕ	$\phi \Rightarrow \psi$	ψ	$+$
`assume φ;`			ϕ	$\neg\phi$	$\bot$	$+$
`assume ¬φ;`			$\neg\phi$	ϕ	$\bot$	$+$
`thus φ by ...;`	ϕ		ϕ	$\phi \wedge \psi$	ψ	$+$
`thus φ by ...;`	ϕ		ϕ	ϕ	$\top$	$+$
`qed by ...;`	ϕ			ϕ		$+$
`take` t`;`				$\exists x \, \psi$	$\psi[x := t]$	$+$
`.=` t_{i+1} `by ...;`	$t_i = t_{i+1}$		$t_1 = t_{i+1}$	ψ	ψ	$-$
`set` x `=` t`;`		x	$x = t$	ψ	ψ	$-$
`consider` x_1 `...` x_n `such that` ϕ `by ...;`	$\exists x_1 \ldots \exists x_n \, \phi$	$x_1 \ldots x_n$	ϕ	ψ	ψ	$-$
`cases by ...;` `...`	$\phi_1 \vee \cdots \vee \phi_n$			ψ		

`suppose` ϕ_i`;`
  `...`
`end;`
`...`
			ϕ_i		ψ	$+$

This then is the basic `miz3` proof language. The `by` justifications should contain sufficient references for the automation to prove the statement in the second column. The third and fourth columns give the variables and statements that are added to the proof context. The fifth and sixth columns give the `thesis` before and after the step, and the seventh column indicates whether the step is a skeleton step or not.

In the case of a `cases`, every `suppose` branch inherits the `thesis` that held at the `cases`, as indicated in the table. However, as one cannot remove a `suppose` without destroying the skeleton of the proof, it still counts as a skeleton step.

The `.=` construction is used for *iterated equalities.* Not indicated in the table (for lack of space) is that the step directly before the `.=` should prove a statement of the shape $t_1 = t_i$. This then determines the terms t_1 and t_i in the table. Using `.=` reasoning, chained equalities of the shape $t_1 = t_2 = t_3 = \cdots = t_n$ can be coded as

```
t1 = t2 by ...;
  .= t3 by ...;
  ...
  .= tn by ...;
```

The `now` syntax can be used when the statement that is being proved can be inferred from the skeleton steps in the proof (which is generally the case). In that situation

$$
\begin{array}{rcl}
& & \overbrace{\hspace{7cm}}^{\text{HOL Light}} \\
\end{array}
$$

⟨*lemma*⟩ ::= (`let` ⟨ident⟩ `=`)$^?$ `thm` `‘;` $\overbrace{\text{⟨formula⟩ ⟨proof⟩} \boxed{;}^? }^{\texttt{miz3}}$ `‘` `;;`

⟨*proof*⟩ ::= ⟨*by refs*⟩

| `proof` ⟨*proof step*⟩* (`end` | `qed` ⟨*by refs*⟩)

⟨*proof step*⟩ ::= ⟨formula⟩ ⟨*labels*⟩ ⟨*proof*⟩ `;`

| `let` ⟨ident⟩$^+$ `be` ⟨type⟩ `;`

| `assume` ⟨formula⟩ ⟨*labels*⟩ `;`

| `thus` ⟨formula⟩ ⟨*labels*⟩ ⟨*proof*⟩ `;`

| `take` ⟨term⟩ `;`

| `consider` ⟨ident⟩$^+$ (`being` ⟨*type*⟩)$^?$ `such` `that`
⟨formula⟩ ⟨*labels*⟩ ⟨*proof*⟩ `;`

| `cases` ⟨*by refs*⟩ `;`
(`suppose` ⟨formula⟩ ⟨*labels*⟩ `;`
⟨*proof step*⟩* (`end` | `qed` ⟨*by refs*⟩) `;`)*

| `now` ⟨*labels*⟩ ⟨*proof step*⟩$^+$ `end` `;`

| `.=` ⟨term⟩ ⟨*labels*⟩ ⟨*proof*⟩ `;`

| `set` ⟨ident⟩ `=` ⟨term⟩ ⟨*labels*⟩ `;`

| `exec` ⟨tactic⟩ `;`

⟨*labels*⟩ ::= (`[` ⟨ident⟩ `]`)*

⟨*by refs*⟩ ::= (`by` ⟨*ref*⟩ (`,` ⟨*ref*⟩)*)$^?$ (`from` ⟨*ref*⟩ (`,` ⟨*ref*⟩)*)$^?$

⟨*ref*⟩ ::= `-` | ⟨ident⟩ | $\underbrace{\text{⟨thm⟩ | ⟨tactic⟩ | ⟨conv⟩}}_{\text{OCaml}}$

Figure 1: The full `miz3` grammar

```
$\phi$ proof ... end;
```

can be abbreviated as

```
now ... end;
```

Finally the `-` label refers to the statement that was introduced most recently to the proof context, and the `exec` step can be used to transform the `thesis` using an arbitrary tactic in the style of [52].

The `miz3` language differs in some respects from the real Mizar language:

- The statements, terms and types use HOL Light syntax instead of Mizar syntax. In particular the rich type system of Mizar [55] is not available.
- The labels are behind the statements in brackets, instead of in front with a colon.
- There is no `then`. The label `-` refers to the previous statement. Also, the last 'horizon' statements are visible without reference, where `horizon` is a variable of the `miz3` server that is usually set to `1`. If one sets `horizon := -1;;` all statements local to the proof that are in scope are used in inference checking without any references. (This makes the proofs look nice and checking very slow.)
- Some keywords are slightly different: see Figure 1 for the exact grammar. This grammar can be parsed with `yacc` without any shift/reduce conflicts. In that sense it is more straightforward than the grammar used by the real Mizar [11]. The most notable changes are that there is a keyword `qed` as an abbreviation of 'thus thesis; end', and that in iterated equalities each step has a terminating semicolon instead of just the last one.
- The `from` keyword has a different meaning than in Mizar (see the explanation below of how tactics in justifications function.)

In a justification of a proof step an arbitrary HOL Light tactic (either of type `tactic` or of type `thm list -> tactic`) can be given. If this tactic is absent, the default tactic `default_prover`, which is another variable of the `miz3` server, is used. This first tries `HOL_BY`, an equivalent of the Mizar automated theorem prover by John Harrison, and if that fails runs some decision procedures, as well as the HOL Light first order automated theorem prover `MESON` [25].

If a tactic is explicitly present in a step, a goalstate is created with the statement that needs to be justified as the goal, and the statements referred to in the `by` part of the justification as the assumptions. In this goalstate the tactic is executed with the `by` part statements as arguments. Finally it is checked whether the subgoals obtained that way are all present in the union of the `by` and `from` parts of the justification. A tactic to connect everything in the style of [52] is executed, which will fail if this is not the case.

4. The implementation of the miz3 proof interface

We now present a low-level description of the implementation of the `miz3` interface. This might seem to be about irrelevant details, but if one leaves out enhancements like the caches and time-outs, the system becomes unusable for serious work. Readers only interested in an abstract description of the `miz3` proof style, or in using the system instead of understanding how it is organized on the inside, can safely skip this section.

The `miz3` interface is both a prototype (in the sense that it has no real users yet) as well as a quite usable system (in the sense that it is a quite usable proof language for the

quite usable HOL Light system). The `miz3` interface is rather light weight, although in its current implementation it needs the infrastructure of a Unix system.

The full source of the `miz3` system consists of five files:

`miz3.ml`	1,903 lines	the OCaml program to be loaded in HOL Light
`miz3_of_hol.ml`	237 lines	(see Section 5 below)
`bin/miz3`	28 lines	the command line batch checker
`bin/miz3f`	40 lines	a filter version of the command line checker
`exrc`	9 lines	the 'vi mode' for `miz3`

The communication between the `vi` session and HOL Light is initiated by `vi` sending a Unix signal to HOL Light, after which the signal handler of the HOL Light session does all the work. This means that the HOL Light session when not doing a `miz3` check is not actively 'waiting' for commands, and that if it is running some other code, then that code will be interrupted to do the `miz3` check. The communication between `vi` and HOL Light is through files, of which some have hard coded special names in order for both sides to be able to find them.

Specifically, when checking a proof from the `vi` interface by typing

$$\boxed{control\text{-}\texttt{C}\ return}$$

the following steps happen:

- The relevant part of the buffer is selected using the `vi` commands `{` and `}`.
- This part of the buffer is filtered through the perl script `miz3f`.
- This script writes this to a temporary file in `/tmp`, and runs the shell script `miz3` on that file.
- This then writes the name of its input file in the file with the special name `/tmp/miz3_filename`, looks up the process id of the HOL Light session running the `miz3` server in `/tmp/miz3_pid`, and sends that process the `USR2` signal.
- The signal handler of the HOL Light session now finds the file it needs to check, and parses it into a data structure of OCaml type `step list`. In this data structure the full input is stored in small pieces.
- Next, the server calls the function `check_proof` on this, which returns another `step list`, this time with errors marked, and possibly grown by running tactics after the `#` symbol.
- This result now is printed back into the file in `/tmp`, after which the filter puts it back into the edit buffer.

In other words the *full* proof is processed *every* time a check is done. To make this acceptably fast, there are two caches. The first cache holds inferences that have already been checked, to prevent the checker from having to run all tactics every time. The second cache holds the OCaml objects associated with the elements in the `by` justifications. These are calculated using the OCaml functions `Toploop.parse_toplevel_phrase` and `Toploop.execute_phrase` (which together are the OCaml equivalent of Lisp's `eval` function).

We do not want to restrict users to 'special' tactics that are designed to always terminate in a reasonable time. This means that while working on a proof sometimes tactics will 'hang'. Therefore, while developing a proof, after a specified time tactics will be killed using the `Unix.alarm` function. This time is given by the variable `timeout`. Of course, this makes it

dependent on the specific computer used whether a proof will be accepted or flagged with a time-out error. (However, a similar thing holds for any interactive theorem prover, because the memory might run out during a check as well.) If one wants to check a finished proof on a slow system without having to worry about time-outs for tactics, one can set `timeout := -1;;` to disable time-outs.

The system will never pretty-print user input by itself, even though it parses and reprints proofs all the time. Even white space and comments will stay exactly the way they are. However, if the proof is modified by 'growing' it through execution of tactics, the system tries hard to indent and line wrap the new parts nicely. There are a dozen parameters to direct this process, which are listed at the start of the source file `miz3.ml`.

Although the interface currently only runs in `vi`, we took great care to not have it be dependent on `vi` specific features. In fact, the whole '`miz3` mode' for `vi` essentially consists of the single line

```
:map ^C<CR> {/.^M!}miz3f^M/#^Ml
```

in the `.exrc` file (for `vim` users the `.vimrc` file), which is the file that configures mappings between `vi` keystrokes and commands. The simplicity of this interface means that porting it to other editors will be trivial.

5. Automatically converting procedural proofs to declarative proofs

Traditional HOL Light proofs can be mimicked in `miz3` by using the system in the procedural style as shown in Section 2. We wrote a small program that automates this process. Using this, *any* proof from HOL Light library can be fully automatically converted to the `miz3` language.

For example, consider the following HOL Light lemma, that says that subtraction on natural and real numbers correspond to each other:

```
let REAL_OF_NUM_SUB = prove
 (`!m n. m <= n ==> (&n - &m = &(n - m))`,
  REPEAT GEN_TAC THEN REWRITE_TAC[LE_EXISTS] THEN
  STRIP_TAC THEN ASM_REWRITE_TAC[ADD_SUB2] THEN
  REWRITE_TAC[GSYM REAL_OF_NUM_ADD] THEN
  ONCE_REWRITE_TAC[REAL_ADD_SYM] THEN
  REWRITE_TAC[real_sub; GSYM REAL_ADD_ASSOC] THEN
  MESON_TAC[REAL_ADD_LINV; REAL_ADD_SYM; REAL_ADD_LID]);;
```

To be able to use the converter, it is necessary to load it:

```
# #use "miz3_of_hol.ml";;
```
⟨*many lines of output*⟩

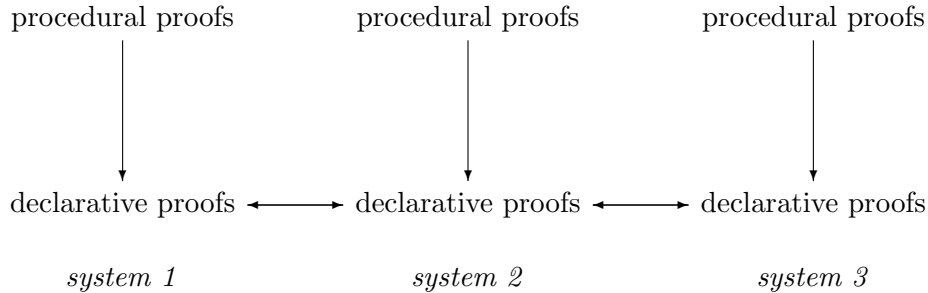Then one converts the proof as follows:

```
# miz3_of_hol "/opt/src/hol_light/real.ml" "REAL_OF_NUM_SUB";;
0..0..1..2..7..14..37..72..174..325..solved at 392
val it : step =
  !m n. m <= n ==> &n - &m = &(n - m) [1]
  proof
    !n m. m <= n ==> &n - &m = &(n - m) [2]
```

```
proof
  let n m be num;
  !d. n = m + d ==> &n - &m = &(n - m) [3]
  proof
    let d be num;
    assume n = m + d [4];
    &d + &m + -- &m = &d [5]
      by MESON_TAC[REAL_ADD_LINV; REAL_ADD_SYM; REAL_ADD_LID],4;
    (&d + &m) - &m = &d [6]
      by REWRITE_TAC[real_sub; GSYM REAL_ADD_ASSOC],4 from 5;
    (&m + &d) - &m = &d [7] by ONCE_REWRITE_TAC[REAL_ADD_SYM],4 from 6;
    &(m + d) - &m = &d [8]
      by REWRITE_TAC[GSYM REAL_OF_NUM_ADD],4 from 7;
  qed by ASM_REWRITE_TAC[ADD_SUB2],4 from 8;
  (?d. n = m + d) ==> &n - &m = &(n - m) [9] by STRIP_TAC from 3;
qed by REWRITE_TAC[LE_EXISTS] from 9;
qed by REPEAT GEN_TAC;
```

The possibility to convert proofs like this does not depend on HOL Light specifics. Any procedural system that has goals and tactics can have a `miz3` layer on top of it and proofs can then be converted to that in exactly the same way. We believe that this gives an approach to 'integrate' mathematical libraries between systems:

procedural proofs   procedural proofs   procedural proofs

| | |
↓   ↓   ↓

declarative proofs ⟷ declarative proofs ⟷ declarative proofs

system 1   system 2   system 3

However, conversions between Mizar style proofs for different systems cannot be completely automatic. Even although the `miz3` language can be put on top of any system, the semantics of the *statements* (when working on top of the native library of the systems) will not exactly match, and the *justification* automation will not match either. When converting a `miz3` proof to Mizar, Isar or C-zar, one can only do an approximate job with the statements, and there is no proper way to translate all tactics.

However, one gets a very good 'starting point' when converting a declarative proof between systems. There just will be some justification errors. This means that a declarative proof converted to a different system will be like a 'formal proof sketch' [53].

For these reasons our system can also not directly make use of the Mizar mathematical library MML. But again, it is not difficult to write a translator that translates a class of Mizar proofs (the ones that use statements that map reasonably well to HOL Light versions) into `miz3` formal proof sketches.

6. A LARGER EXAMPLE: LAGRANGE'S THEOREM

We have tried `miz3` on a somewhat larger proof: Lagrange's theorem from group theory. This states that for any group the order of a subgroup always divides the order of that group. John Harrison already had written a HOL Light proof of this theorem, which meant that we could see how the traditional proof style compared to what was possible in `miz3`. We used two different approaches: we wrote a proof following the proof that is in van der Waerden's book about algebra [48], and we wrote a `miz3` proof trying to closely follow John Harrison's proof.

Van der Waerden's proof we formalized both in Mizar and in `miz3`, and in both cases we first wrote a formal proof sketch [53] of the proof. The `miz3` formal proof sketch was:

```
now let a be A; assume a IN G; let b be A; assume b IN G;
  assume i(a)**b IN H;
  b***H = a**i(a)**b***H; .= a***(i(a)**b***H); thus .= a***H;
end;
!a b. a IN G /\ b IN G /\ ~(a***H = b***H) ==> a***H INTER b***H = {}
proof let a be A; assume a IN G; let b be A; assume b IN G;
  now assume ~(a***H INTER b***H = {});
    consider g1 g2 such that g1 IN H /\ g2 IN H /\ a**g1 = b**g2;
    g1**i(g2) = i(a)**b;
    i(a)**b IN H;
    thus a***H = b***H;
  end;
qed;
!a. a IN G ==> a IN a***H proof let a be A; assume a IN G; a**e = a; qed;
{a***H | a IN G} PARTITIONS G;
!a b. a IN G /\ b IN G ==> CARD (a***H) = CARD (b***H)
proof let a be A; assume a IN G; let b be A; assume b IN G;
  consider f such that !g. g IN H ==> f(a**g) = b**g;
  bijection f (a***H) (b***H);
qed;
set INDEX = CARD {a***H | a IN G};
set N = CARD G; set n = CARD H; set j = INDEX; N = j*n;
thus CARD H divides CARD G;
```

(The notations with the multiple stars is a bit ugly, but we wanted to use the statement from John Harrison's proof, and there group multiplication is written as `**`. We then used `***` for multiplication of an element and a coset.) Both the Mizar and `miz3` formalizations were completely straightforward. The part of the final formalization that corresponds to the first four lines of the formal proof sketch became:

```
now [22]
  let a be A; assume a IN G [23];
  let b be A; assume b IN G [24];
  i(a)**b IN G [25] by 2,23,24;
  assume i(a)**b IN H [26];
  b***H = e**b***H by 2,24;
    .= a**i(a)**b***H by -,2,23;
    .= a**(i(a)**b)***H by -,2,23,24;
    .= a***(i(a)**b***H) by -,9,23,25;
```

```
  thus .= a***H by -,17,26;
end;
```

We obtained `miz3` versions of John Harrison's proof in two different ways. We automatically generated this using the technology from Section 5, but we also wrote one manually by just running the proof step by step, 'understanding' what was going on, and then rendering that in `miz3` syntax. The first, although a correct declarative proof, was disappointingly large, because there is a lot of equality reasoning in the proof and the converter does not optimize that proof pattern yet. Writing the second again was completely straightforward.

The line counts of the various formalizations that we got were:

traditional HOL Light by John Harrison	**214** lines
Mizar formal proof sketch	25 lines
Mizar	153 lines
`miz3` formal proof sketch	23 lines
`miz3`	**183** lines
`miz3` (converted from John Harrison's proof)	1,317 lines
`miz3` (manually written following John Harrison's proof)	**198** lines

It is clear that the two hand-written `miz3` formalizations are of a similar size to the HOL Light and Mizar formalizations, which shows that using `miz3` did not lead to much larger proof texts.

7. Discussion

7.1. **A more generic interface?** The synthesis between proof styles that we propose in this paper and its accompanying proof language is very general. Therefore a natural question is why we did not make the `miz3` framework *generic*, like for example the Proof General interface [6]. That way our work would be useful for users of systems like Isabelle, Coq and PVS.

The reason we did not do this is that to get a usable framework we had to do various things that tie deep into the innards of the system. Our current implementation really works on the level of the data structures inside the prover. And there we do things like timing out tactics, caching justifications, and interpreting strings as OCaml expressions.

To make a version of our system that is generic, we would need to work on a much more syntactic level. However, in that case it probably would not be so easy to do caching of justifications in a sound way. We still think this might be possible, but it would be very different from the architecture that we use now.

7.2. **The reliability of the system.** Since we have not changed the HOL Light kernel, `miz3` is as reliable as the standard version of HOL Light. If `miz3` gives an error message, then *that* might be wrong, but if a proof is *accepted*, then we know that the kernel of the system has fully checked the proof object, and therefore that possible `miz3` bugs did not matter (this is called the de Bruijn criterion [8]). This holds even although `miz3` consists of complicated code, and even although the system goes outside the basic OCaml programming model by using a system call to time out tactics and by invoking the OCaml interpreter on pieces of the proof text.

When a lemma has a completed `miz3` proof with no errors left, the proved `thm` will 'appear' in the session for further use. For example, once we check a correct proof for the example lemma in Section 2, we see magically appear in the HOL Light session:

```
# val ( ARITHMETIC_SUM ) : thm =
  |- !n. nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2
```

From then on we can refer to the variable `ARITHMETIC_SUM`. The interface generally suppresses all printing when it is doing a check (we do not want a check from `vi` to disturb the output of our HOL Light session, where we might be doing other things), only when a proof is fully correct the theorem is printed.

To make it possible to get a `thm` out of a `miz3` proof, the justification cache has not only have to hold the information that a justification was correct, but also a `thm` that can be used to redo that justification in a very fast way.

7.3. **Future work.** The `miz3` interface is working very well, and is competitive with the traditional way of using HOL Light. Still, there are various ways in which it can be improved:

- The default prover for `by` justifications is not completely satisfactory. This is orthogonal to the design of the system, but a better justifier will make the system easier to use. One would like the justifier to have the following properties:
  - Any statement that can be proved in HOL Light should be provable in `miz3` without explicit tactics. This does not mean that the `by` justifier should be able to prove it all by itself (not even given infinite time and memory), but that it should be possible to break the proof in sufficiently small steps that `by` then can prove each *step* without further procedural help.
  - Basic HOL Light tactics like `REWRITE_TAC`, `MATCH_MP_TAC`, `ARITH_TAC` and `MESON_TAC` should not have to be given explicitly. If the tactic in a justification is one of these, and it runs in the justification in a reasonably short time, then the default tactic should be able to do the proof too.
- Experimenting with goal states that correspond to steps in a `miz3` proof could be more ergonomic. These goal states either can correspond to a `thesis` at a specific place in the proof, or to a `by` justification. One can experiment already using the `GOAL_TAC` tactic (a custom `miz3` tactic that sets the current goal of the HOL Light session to the goalstate it is given as input) but this is a bit cumbersome. At the moment Isabelle/Isar is in this respect much more ergonomic than our system.

  Currently, when the system flags an error for a certain place in the proof this gives the user a bit of a helpless feeling (the same holds for the Mizar system). If the user understands the problem, then all is fine, but if the user does *not* understand the error then there should be something that can be done to investigate more easily than is possible now.
- We might try going beyond the standard Mizar proof idiom. For instance, it would be trivial (although not terribly useful) to follow the suggestion from various people to have an *it now is sufficient to prove this* proof step. Such a step sets the `thesis` and its justification derives the old `thesis` from that statement. It is called '`suffices to show`' in [24] and '`Suffices`' in [7].

  More interestingly there could be a variant of the `cases` construct where the cases are the subgoals produced by a tactic. That way induction proofs could be more

like traditional mathematical proofs, because then the `INDUCT_TAC` tactic would be *before* the inductive cases.

- It would be interesting to have a more intelligent folding editor on top of our interface. It could color the relevant parts of the proof to clearly show the goal that corresponds to an unjustified step, and it could intelligently fold and unfold subproofs.
- It would be useful to make the communication architecture more flexible. That way we might have verification not be restricted to one contiguous block, the system could report errors more precisely, and several instances of the system could be run on one computer at the same time.

7.4. **A proof interface for the working mathematician.** A good proof interface for formal mathematics should satisfy the requirement that *easy things should be easy, and hard things should be possible* [9]. The synthesis proposed in this paper combines the ease of the declarative proof style with the power of the procedural proof style. We hope that our approach will turn out to be a piece in the puzzle of making interactive theorem provers useful for and attractive to the working mathematician.

Acknowledgements

References

[1] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] Agda Development Team. The Agda Wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`.

[3] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Crafting a proof assistant. In *Proceedings of the 2006 international conference on Types for proofs and programs*, TYPES'06, pages 18–32, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Andrea Asperti et al. Mathematical Knowledge Management in HELM. *Annals of Mathematics and Artificial Intelligence*, 38(1-3):27–46, 2003.

[5] Andrea Asperti and Bernd Wegner. MOWGLI – A New Approach for the Content Description in Digital Documents. In *Proc. of the 9th Intl. Conference on Electronic Resources and the Social Role of Libraries in the Future*, volume 1, Autonomous Republic of Crimea, Ukraine, 2002.

[6] David Aspinall. Proof General: A generic tool for proof development. In *European Joint Conferences on Theory and Practice of Software (ETAPS)*, 2000.

[7] Henk Barendregt. Towards an Interactive Mathematical Proof Language. In F. Kamareddine, editor, *Thirty Five Years of Automath*, pages 25–36. Kluwer, 2003.

[8] Henk Barendregt and Freek Wiedijk. The Challenge of Computer Mathematics. *Transactions A of the Royal Society*, 363(1835):2351–2375, 2006.

[9] Douglas Beaver et al. Amazon.com Interview: Larry Wall. The father of Perl talks about XML, Unicode, the Win32 port, and the philosophy behind the language, `http://www.amazon.com/gp/feature.html?docId=7137`, 1998.

[10] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, and D. Vasaru. An Overview on the Theorema project. In W. Kuechlin, editor, *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation)*, Maui, Hawaii, 1997. ACM Press.

[11] Paul Cairns and Jeremy Gow. Using and Parsing the Mizar Language. *Electronic Notes in Theoretical Computer Science*, 93:60–69, 2004.

[12] Robert L. Constable, Stuart F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, Douglas J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[13] Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2010.

[14] Pierre Corbineau. A declarative proof language for the Coq proof assistant. In *TYPES 2007 : Types for Proofs and Programs*, number 4941 in LNCS. Springer-Verlag, 2007.

[15] N.G. de Bruijn. The Mathematical Vernacular, a language for mathematics with typed sets. In P. Dybjer et al., editors, *Proceedings of the Workshop on Programming Languages*, Marstrand, Sweden, 1987.

[16] Anthony Fox. Formal Specification and Verification of ARM6. In D.A. Basin and B. Wolff, editors, *TPHOLs 2003*, volume 2758 of *LNCS*. Springer, 2003.

[17] Georges Gonthier. A computer-checked proof of the Four Colour Theorem. `http://research.microsoft.com/~gonthier/4colproof.pdf`, 2006.

[18] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008.

[19] Mike Gordon and Tom Melham, editors. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993.

[20] Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz. Mizar in a Nutshell. *Journal of Formalized Reasoning*, 3:153–245, 2010.

[21] Frédérique Guilhot. Proof explanations: using natural language and graph view, 2003. presentation for the MoWGLI project.

[22] Tom Hales. Mathematics in the Age of the Turing Machine, 2011. To appear in 2012 in a special volume of Lecture Notes in Logic (CUP) in commemoration of the centennial of Turing's birth.

[23] John Harrison. The HOL Light theorem prover. `http://www.cl.cam.ac.uk/~jrh13/hol-light/`.

[24] John Harrison. A Mizar Mode for HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *TPHOLs '96*, volume 1125 of *LNCS*, 1996.

[25] John Harrison. Optimizing Proof Search in Model Elimination. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction*, volume 1104 of *LNCS*, pages 313–327, New Brunswick, NJ, 1996. Springer-Verlag.

[26] John Harrison. Proof Style. In Eduardo Giménez and Christine Paulin-Möhring, editors, *Types for Proofs and Programs: International Workshop TYPES'96*, volume 1512 of *LNCS*, pages 154–172, Aussois, France, 1996. Springer-Verlag.

[27] John Harrison. *The HOL Light manual (1.1)*, 2000.

[28] John Harrison. *HOL Light Tutorial (for version 2.20)*, 2007.

[29] John Harrison. Formalizing an Analytic Proof of the Prime Number Theorem. In R. Boulton, J. Hurd, and K. Slind, editors, *Tools and Techniques for Verification of System Infrastructure*, pages 243–261. The Royal Society, 2008.

[30] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.

[31] Cezary Kaliszyk and Freek Wiedijk. Merging procedural and declarative proof. In S. Berardi, F. Damiani, and U. De'Liquoro, editors, *TYPES 2008*, 2009.

[32] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, 2000.

[33] Gerwin Klein et al. seL4: formal verification of an OS kernel. In J.N. Matthews and Th. Anderson, editors, *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[34] Xavier Leroy. Formal Certification of a Compiler Back-end, or: Programming a Compiler with a Proof Assistant. In *POPL'06*, 2006.

[35] Lemma 1 Ltd. *ProofPower – Description*. Lemma 1 Ltd., 2000.

[36] Conor McBride and James McKinna. The View from the Left. *Journal of Functional Programming*, 14(1), 2004.

[37] Norman D. Megill. Metamath, A Computer Language for Pure Mathematics. `http://us.metamath.org/downloads/metamath.pdf`, 1997.

[38] Tobias Nipkow, Larry Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[39] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752, Berlin, Heidelberg, New York, 1992. Springer.

[40] Andrei Paskevich. The syntax and semantics of the ForTheL language. `http://nevidal.org/download/forthel.pdf`.

[41] Jeff Pelletier. A History of Natural Deduction and Elementary Logic Textbooks. In J. Woods and B. Brown, editors, *Logical Consequence: Rival Approaches and New Studies, Vol. 1*. Hermes Science Pubs., 2000.

[42] Frank Pfenning and Carsten Schuermann. *Twelf User's Guide, Version 1.4*, 2002. `http://www-2.cs.cmu.edu/~twelf/guide-1-4/twelf.ps`.

[43] Randy Pollack. How to Believe a Machine-Checked Proof. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, Oxford, 1998.

[44] Christophe Raffalli. The PhoX Proof Assistant. `http://www.lama.univ-savoie.fr/~RAFFALLI/phox.html`.

[45] Claudio Sacerdoti Coen. Declarative Representation of Proof Terms. *J. Autom. Reason.*, 44:25–52, 2010.

[46] Raymond Smullyan. *What Is the Name of this Book?* Penguin Books Ltd, 1990.

[47] Josef Urban. MizarMode - an integrated proof assistance tool for the Mizar way of formalizing mathematics. *J. Applied Logic*, 4(4):414–427, 2006.

[48] B.L. van der Waerden. *Algebra I*. Springer-Verlag, 1971. Achte Auflage.

[49] Markus Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002. `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html`.

[50] Markus Wenzel. *The Isabelle/Isar Reference Manual*. TU München, 2002.

[51] Markus Wenzel and Freek Wiedijk. A comparison of the mathematical proof languages Mizar and Isar. *Journal of Automated Reasoning*, 29:389–411, 2002.

[52] Freek Wiedijk. Mizar Light for HOL Light. In R. J. Boulton and P. B. Jackson, editors, *TPHOLs 2001*, volume 2152 of *LNCS*, 2001.

[53] Freek Wiedijk. Formal Proof Sketches. In S. Berardi, M. Coppo, and F. Damiani, editors, *TYPES 2003*, volume 3085 of *LNCS*, pages 378–393, 2004.

[54] Freek Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *LNCS*. Springer, 2006. With a foreword by Dana S. Scott.

[55] Freek Wiedijk. Mizar's Soft Type System. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics 2007*, volume 4732 of *LNCS*, pages 383–399, 2007.