Mizar Light for HOL Light

Freek Wiedijk

freek@cs.kun.nl
Department of Computer Science
University of Nijmegen
The Netherlands

Abstract. There are two different approaches to formalizing proofs in a computer: the procedural approach (which is the one of the HOL system) and the declarative approach (which is the one of the Mizar system). Most provers are procedural. However declarative proofs are much closer in style to informal mathematical reasoning than procedural ones.
There have been attempts to put declarative interfaces on top of procedural proof assistants, like John Harrison's Mizar mode for HOL and Markus Wenzel's Isar mode for Isabelle. However in those cases the declarative assistant is a *layer* on top of the procedural basis, having a separate syntax and a different 'feel' from the underlying system.
This paper shows that the procedural and the declarative ways of proving are related and can be integrated seamlessly. It presents an implementation of the Mizar proof language on top of HOL that consists of only 41 lines of ML. This shows how close the procedural and declarative styles of proving really are.

1 Introduction

We describe a programming experiment with the HOL system. To be able to read this paper one has to have some familiarity with both the HOL [10] and Mizar [12, 17] systems. The software described here is not meant to be used for serious work (and it certainly doesn't emulate the full Mizar language). Rather it's a kind of 'thought experiment' to clarify the relation between the procedural and declarative styles of proving.

This paper uses HOL for the procedural prover. However the way the paper integrates Mizar-style proofs with it also applies to other procedural systems like Coq [2] and Isabelle [14]. For each of these systems a set of 'Mizar tactics' could be written as described in this paper, giving them 'Mizar style' declarative proofs without a separate syntactic layer.

The plan of this paper is as follows. It first presents the procedural and declarative styles of proving and the HOL and Mizar systems. Then Section 6 presents the main idea of the paper which is implementing Mizar steps as HOL tactics. For each of these 'Mizar tactics' this section gives an ML type in the framework of the HOL system. After that the paper discusses various variations on this Mizar mode for HOL. The paper concludes with a bigger example and an outlook. The source code of the Mizar mode is an appendix.

2 Procedural versus declarative proving

The idea of formalizing mathematical proofs in such a way that a computer can check the correctness is not new but only recently it has become practical and popular. Most of the proofs that are currently checked are proofs of the correctness of computer software or hardware but some people have also been formalizing other kinds of proofs (for instance of mathematical theorems [3, 6]).

There are two main styles of proof checking programs (the so-called proof checkers or proof assistants): the *procedural* style and the *declarative* style.

The procedural proof checkers descend from a system from the seventies called LCF [8]. Such a system has a *proof state* which consists of a set of 'proof obligations'. This state is transformed by means of so-called *tactics* which take a proof obligation and reduce it to several simpler ones (possibly none). The proof process starts with the statement to be proved as the sole proof obligation; once no proof obligations remain, the proof is complete. Procedural proofs consisting of a sequence of tactics cannot be understood without interactively running them on a computer because they only contain the transitions between the proof states and not the proof states themselves.[1] Also since the initial proof obligation is the final statement to be proved, procedural proofs tend to run *backwards*, from the conclusion back to the assumptions.

The other two proof checkers from the seventies, Automath [13] and Mizar [12, 17], both are of the declarative kind. (Another system that is declarative is the Ontic system by David McAllester [11].) In a declarative system, a proof doesn't consist of instructions to transform statements but of those statements themselves. Furthermore, the statements are present in deductive order: the assumptions are at the start and the conclusion is at the end.[2]

The procedural and declarative styles differ in several ways:

- Declarative proofs are closer to informal mathematical reasoning and therefore are more readable than procedural proofs because in a declarative proof the statements are present themselves and in the proper order.
- Most current proof assistants are procedural (perhaps because LCF style provers are programmable by their users, while declarative systems can generally only be extended by its developers).
- Declarative proofs can be written without the proof assistant running and then checked with a batch oriented system. Procedural proofs can only be developed interactively because the computer has to keep track of the proof obligations. Indeed, some declarative provers only have a batch mode while procedural provers always also have an interactive mode.

[1] Procedural proofs are therefore sometimes presented using *runs* of proof scripts (or, even better, proof trees) showing intermediate goals.

[2] The relation between backward and forward reasoning as related to procedural and declarative proofs is more subtle than is suggested here. Declarative proofs (and informal mathematical reasoning as well) also take backward steps. And many procedural provers also can do forward reasoning. However the particular tactic collections found in procedural provers tend to be biased towards backward reasoning.

- Since a declarative proof contains a chain of statements, it is fairly robust with respect to errors. If one step in the proof is erroneous, a declarative system can recover from the error and check the rest of the proof file reasonably well. In contrast, a procedural prover stops checking at the first error it encounters. So after the point in the file where an error occurs, a procedural system can't say much about correctness.

It seems natural to look for a way to integrate the procedural and declarative approaches to proving. Two attempts to put a declarative interface on a procedural prover are the Mizar mode for HOL by John Harrison [9] and the Isar mode for Isabelle by Markus Wenzel [16].

The Mizar mode for HOL by John Harrison is a true integration, in the sense that the Mizar commands behave like ordinary HOL tactics. However this doesn't become clear from the way this mode is presented. For instance, the Mizar sub-language has a separate parser from the (ML based) HOL proof language. Also once the Mizar mode is active, the normal parser of the HOL terms is no longer easily available. This seems to suggest that once the Mizar mode has been activated, the 'normal' style of procedural HOL proving is not meant to be used anymore.

The Mizar mode for HOL in this paper is a variation on the Mizar mode of John Harrison. Its main difference with Harrison's Mizar mode is that the Mizar primitives *are* HOL tactics (so are not just compiled to them) and therefore the integration is very tight. Also the implementation of our Mizar mode takes only 41 lines of ML, which is smaller than Harrison's implementation which consists of about 650 lines of ML.

The Isar mode for Isabelle differs from the Mizar mode for HOL in that it has outgrown the experimental stage and has been used for serious proofs [3, 5, 4]. However it really is a second layer on top of the Isabelle layer (although it is possible to 'embed' tactic scripts in Isar proofs), so in this case there is no mixture between the two approaches. In fact, both layers have their own proof state (called 'the static proof context' and 'the dynamic goal state') which are to be synchronized at certain checkpoints. This is presented as a benefit because it makes it possible to give a different order to the proof steps from the order imposed by the underlying procedural basis but it shows that there is no tight integration.

Two other declarative systems that integrate declarative proofs with higher order logic are the SPL system by Vincent Zammit [18] and the Declare system by Don Syme [15]. These two systems are not meant as a declarative interface to a procedural prover. Instead they are autonomous declarative systems. (The SPL system is *implemented* on top of the HOL system but it is not an interface to HOL.)

Procedural and declarative proof checkers both might or might not satisfy what Henk Barendregt calls *the de Bruijn principle* and *the Poincaré principle* [1]. This paper is about the relation between procedural and declarative proof styles. This is unrelated to the issue of whether a system should satisfy either of these principles or how to make it do so.

3 Example: the drinker

As a running example in this paper, we will use the so-called *drinker's principle.* This says that in every group of people one can point to one person in the group such that if that person drinks then all the people in the group drink. This somewhat surprising statement becomes in first order predicate logic:

$$\exists x. \big(P(x) \to \forall y. P(y)\big)$$

The HOL version of this is:

```
?x:A. P x ==> !y. P y
```

(so in HOL '`?`' is the existential quantifier and '`!`' is the universal quantifier) and the Mizar version is:

```
ex x st P x implies for y holds P y
```

Here is an informal proof of the drinker's principle (we will see below how this textual version compares both to the proofs of this statement in the HOL and Mizar systems):

> *Suppose that $P(x)$ is false for some x. Then for that x the implication holds because from a false proposition one may deduce anything. That means that in this case the proposition follows.*
> *Now suppose that $P(x)$ is false for no x. Then $P(x)$ is true for all x. But that statement is the conclusion of the implication and so the proposition again follows.*

Almost all example proofs in this paper are proofs of this simple statement. In Section 8 we will present a bigger example.

4 HOL

The HOL system [7] by Mike Gordon is a descendant from LCF that implements Church's *Higher Order Logic* (hence the acronym 'HOL'), which is a classical higher order logic encoded by simply typed lambda calculus with ML style polymorphism. The system guarantees the correctness of its proofs by reducing everything in LCF style to a 'proof kernel', which because of its small size (nine pages of ML code, about half of which is comment) can be thoroughly checked for correctness by inspection.

The HOL system has had several implementations: HOL88, HOL90, HOL98, ProofPower (a commercial version) and HOL Light. The HOL Light system [10] which is the HOL re-implementation by John Harrison, is the version of the system that we have used for this paper (but all versions are similar). It has been written in CAML Light and consists of slightly under two megabytes of ML source, which implement a mathematical framework containing a formalization of the real numbers, analysis and transcendental functions and several decision procedures for both logical and arithmetical problems. It has been both used for computer science applications and for formalizing pure mathematics (like the fundamental theorem of algebra).

The drinker's principle can be proved in HOL in the following way:

```
let DRINKER = prove
 (`?x:A. P x ==> !y. P y`,
  ASM_CASES_TAC `?x:A. ~P x` THEN
  RULE_ASSUM_TAC (REWRITE_RULE[NOT_EXISTS_THM]) THENL
  [POP_ASSUM CHOOSE_TAC; ALL_TAC] THEN
  EXISTS_TAC `x:A` THEN
  ASM_REWRITE_TAC[]);;
```

There are various ways to prove this statement, but this tactic sequence is a fairly normal way to prove something like this in HOL.

5 Mizar

The Mizar system [12, 17] by Andrzej Trybulec and his group in Bialystok, Poland, is a declarative prover that goes back to the seventies. It implements classical first order logic with ZFC-style set theory. The current version, called PC Mizar, dates from 1989. It consists of a suite of Pascal programs which are distributed compiled for Intel processors (both Windows and Linux). These programs are accompanied by a huge library of all kinds of mathematics, in the form of a series of 686 so-called 'articles' which together are about 1.3 million lines of Mizar.

In Mizar a proof of the drinker's principle looks like:

```
ex x st P x implies for y holds P y
proof
  per cases;
  suppose A0: ex x st not P x;
    consider a such that A1: not P a by A0;
    take a;
    assume A2: P a;
    A3: contradiction by A1,A2;
    thus A4: for y holds P y by A3;
  suppose A5: for x holds P x;
    consider a such that A6: not contradiction;
    take a;
    thus A7: P a implies for y holds P y by A5;
end;
```

Note that this is readable and similar to the informal proof in Section 3.

The Mizar system has many interesting ideas. For instance it has a complicated type system with polymorphic types, overloading, subtyping and type modifiers called 'adjectives', together with powerful type inference rules. Also it has a mathematical looking operator syntax with not only prefix and infix operators but also 'aroundfix' operators, which behave like brackets. However in this paper we will restrict ourselves to the proof language of Mizar. That means that from now on we will only have HOL types and HOL term syntax and we will not mix those with Mizar types and Mizar term syntax. The same restriction to just the proof fragment of Mizar was chosen for the Mizar mode for HOL by John Harrison.

The reasoning part of Mizar turns out to be simple. In its basic form it is given by the following context free grammar:

proposition = [*label* `:`] *formula*

statement = *proposition justification*

justification =
  empty
| `by` *label* { `,` *label*}
| `proof` {*step*} [*cases*] `end`

step =
  statement `;`
| `assume` *proposition* `;`
| `consider` *variable* { `,` *variable*}
    `such that` *proposition* {`and` *proposition*} *justification* `;`
| `let` *variable* { `,` *variable*} `;`
| `take` *term* { `,` *term*} `;`
| `thus` *statement* `;`

cases = `per cases` *justification* `;` {`suppose` *proposition* `;` {*step*}}

empty =

The main Mizar proof feature that is missing from this language fragment is the use of '`.=`' for equational reasoning.

Note that this grammar has only seven kinds of proof elements: a statement without keyword, `assume`, `consider`, `let`, `per cases`, `take` and `thus`. (Compare this with the hundreds of different tactics, tacticals, conversions and conversionals that appear all over the HOL proofs.)

6 Mizar as HOL tactics

We will compare the way the Mizar and HOL systems implement natural deduction. That will lead to a natural ML type (in the framework of HOL) for each of the Mizar steps. The table that lists these types is the essence of our Mizar implementation on top of HOL. (The appendix will show how to implement the Mizar steps according to these types.)

There are two kinds of Mizar steps:

- *skeleton* steps: the natural deduction way of reasoning

– *forward* steps: statements that get added to the context after having been justified with the '`by`' justification

The natural deduction rules correspond to Mizar steps according to following table (rules for which a '–' appears in this table are implemented as forward steps and don't have a Mizar step of their own):

natural deduction	*Mizar*
$\rightarrow$ introduction	`assume`
$\rightarrow$ elimination	–
$\wedge$ introduction	`thus`
$\wedge$ elimination	–
$\vee$ introduction	–
$\vee$ elimination	`per cases`
$\forall$ introduction	`let`
$\forall$ elimination	–
$\exists$ introduction	`take`
$\exists$ elimination	`consider`

The HOL language has natural deduction as well. The main difference is that the Mizar steps make the propositions explicit that the HOL steps leave implicit.

Compare the two ways to do $\rightarrow$ introduction. Suppose we want to reduce the goal:

$$A, B \vdash (C \rightarrow D) \rightarrow E$$

to:

$$A, B,\ (C \rightarrow D) \vdash E$$

(this is the *introduction* rule because a goal oriented prover reasons backwards). The Mizar step doing this is:

```
assume A2: C implies D;
```

(here '`A2`' is the label of the assumption). The same is accomplished in HOL by:

```
DISCH_TAC
```

or if we write out the proof state transformation explicitly:

```
it : goalstack = 1 subgoal (1 total)

 0 [`A`]
 1 [`B`]

`(C ==> D) ==> E`

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

 0 [`A`]
 1 [`B`]
 2 [`C ==> D`]

`E`
```

The Mizar statement gives the 'redundant' information what the discharged statement is and where it ends up in the context. We can imitate this in HOL by defining a tactic `ASSUME_A` such that the HOL tactic becomes:

```
ASSUME_A(2,`C ==> D`)
```

or explicitly:

```
it : goalstack = 1 subgoal (1 total)

 0 [`A`]
 1 [`B`]

`(C ==> D) ==> E`

#e (ASSUME_A(2,`C ==> D`));;
it : goalstack = 1 subgoal (1 total)

 0 [`A`]
 1 [`B`]
 2 [`C ==> D`]

`E`
```

All that the `ASSUME_A` tactic does is check that the number and statement fit and then apply `DISCH_TAC`.

The `ASSUME_A` tactic has the type:

`int` × `term` → `tactic`

If we continue along this line of thought, it turns out that every Mizar construction has a 'natural' HOL type. The table that gives these types is the essence of our Mizar mode for HOL:

`A`	`int` × `term` → `tactic` → `tactic`
`ASSUME_A`	`int` × `term` → `tactic`
`BY`	`int list` → `thm list` → `tactic`
`CONSIDER`	`term list` → `(int` × `term) list` → `tactic` → `tactic`
`LET`	`term list` → `tactic`
`PER_CASES`	`tactic` → `((int` × `term)` × `tactic) list` → `tactic`
`TAKE`	`term list` → `tactic`
`THUS_A`	`int` × `term` → `tactic` → `tactic`

Note that this table corresponds to the Mizar proof grammar from Section 5. Implementing these eight tactics is trivial, as shown in the appendix. The first of these tactics, the `A` tactic, corresponds to a Mizar step without a keyword: a forward reasoning step. The `BY` tactic is used to justify steps. It takes two lists of arguments. The first list is a list of integers referring to the assumption list of the current goal, the second list is a list of `thm`s.

Now we can write down the proof of the drinker's principle as a normal HOL proof (so with `prove` and a chain of tactics put together with `THEN`), this time using the 'Mizar tactics':

```
let DRINKER = prove
 (`?x:A. P x ==> !y. P y`,
  PER_CASES (BY [][])
  [(0,`?x:A. ~P x`),
   (CONSIDER [`a:A`] [(1,`~(P:A->bool) a`)] (BY [0][]) THEN
    TAKE [`a:A`] THEN
    ASSUME_A(2,`(P:A->bool) a`) THEN
    A(3,`F`) (BY [1;2][]) THEN
    THUS_A(4,`!y:A. P y`) (BY [3][]))
  ;(0,`!x:A. P x`),
   (CONSIDER [`a:A`] [] (BY [][]) THEN
    TAKE [`a:A`] THEN
    THUS_A(1,`P a ==> !y:A. P y`) (BY [0][]))]);;
```

Note that this is similar to the Mizar version of this proof from Section 5. The main difference is that type annotations are needed (`` `(P:A->bool) a` `` instead of `` `P a` ``). This problem of having to put type annotations in terms is standard in HOL. A possible approach to this problem will be presented in Section 7.5.

7 Enhancements

The Mizar tactics that we presented in the previous section are very basic. They can be enhanced in various ways. Because we lack the space we don't give all the details. Curious readers are referred to the HOL Light source file `miz.ml` at the URL `<http://www.cs.kun.nl/~freek/mizar/miz.ml>`.

7.1 The `BECAUSE` tactic

The common way to justify a Mizar step in our Mizar mode is with a justification that looks like:

`(BY [`*local statement list*`][`*global statement list*`])`

This has the prover 'hardwired in' (in the implementation from the appendix, this prover first tries `REWRITE_TAC` and if that fails tries `MESON_TAC`). However the `BY` tactic is built on top of a tactic called '`BECAUSE`' which has the type:

`(thm list → tactic) → int list → thm list → tactic`

This means that one can use it with any tactic that has the type `thm list → tactic` like `REWRITE_TAC`, `SIMP_TAC`, `MESON_TAC` and so on. (One also could use it with tactics like `ASM_REWRITE_TAC` but that would be silly, because the assumptions already are accessible through the 'local statements' argument.) For instance one could justify a step by:

`(BECAUSE ONCE_SIMP_TAC[`*local statement list*`][`*global statement list*`])`

This would prove the statement being justified using the `ONCE_SIMP_TAC` tactic with the relevant `thms`.

The `BECAUSE` tactic gives control over the exact behavior of the prover if that is needed, making a refined version of `BY` unnecessary.

7.2 A more powerful ASSUME_A tactic

The ASSUME_A tactic is the declarative version of the procedural DISCH_TAC tactic. The implementation from the appendix mirrors DISCH_TAC exactly. However since the ASSUME_A tactic contains the statement that is discharged, it can be more general. It becomes more powerful if we implement it as:

```
let ASSUME_A (n,tm) =
  DISJ_CASES_THEN2
    (fun th -> REWRITE_TAC[th] THEN N_ASSUME_TAC n th)
    (fun th -> REWRITE_TAC[REWRITE_RULE[] th] THEN
      MIZ_ERROR_TAC "ASSUME_A" [n])
    (SPEC tm EXCLUDED_MIDDLE);;
```

For instance in that case one can use it to reason by contradiction:

```
it : goalstack = 1 subgoal (1 total)

`A`

#e (ASSUME_A(0,`~A`));;
it : goalstack = 1 subgoal (1 total)

 0 [`~A`]

`F`
```

This is also how the assume step of the real Mizar system behaves.

7.3 An interactive version of the PER_CASES tactic

The PER_CASES tactic has the type:

$$\texttt{PER\_CASES} : \texttt{tactic} \to ((\texttt{int} \times \texttt{term}) \times \texttt{tactic})\ \texttt{list} \to \texttt{tactic}$$

In order to debug a proof interactively a version that has type:

$$\texttt{PER\_CASES} : \texttt{tactic} \to (\texttt{int} \times \texttt{term})\ \texttt{list} \to \texttt{tactic}$$

and that leaves the cases as subgoals is more practical. Using this variant makes the proof look less like Mizar because the list of the statements of the cases has been separated from the list of proofs of the cases.

A hybrid is also possible that has the type of the original PER_CASES but doesn't require all the cases to be completely proved after the tactic finishes.

7.4 Tactics versus proofs

Some of the tactic arguments of the Mizar tactics not only have to *reduce* the proof obligations but they have to prove the goal altogether. So those arguments are more 'proofs' than 'tactics'. One might try to reflect this in the typing of the Mizar tactics by at certain places changing tactic (which is defined as goal $\to$ goalstate) to goal $\to$ thm. The second type is in a way a special case ('subtype') of the first. Then functions:

$$\texttt{PROOF} : \texttt{tactic} \to \texttt{goal} \to \texttt{thm}$$
$$\texttt{PER} : (\texttt{goal} \to \texttt{thm}) \to \texttt{tactic}$$

map these two types to each other and:

$$\texttt{prove'} : \texttt{term} \times (\texttt{goal} \to \texttt{thm}) \to \texttt{thm}$$

runs a proof. Using this approach, the Mizar mode tactics will get the following type assignments:

A	int × term → (goal → thm) → tactic
ASSUME_A	int × term → tactic
BY	int list → thm list → goal → thm
CASES	(goal → thm) → ((int × term) × (goal → thm)) list → goal → thm
CONSIDER	term list → (int × term) list → (goal → thm) → tactic
LET	term list → tactic
TAKE	term list → tactic
THUS_A	int × term → (goal → thm) → tactic

(Note that we have separated the `PER_CASES` tactic into a combination of `PER` and `CASES`.) The example proof becomes, using these tactics:

```
let DRINKER = prove'
 (`?x:A. P x ==> !y. P y`,
  PROOF
  (PER (CASES (BY [][])
    [(0,`?x:A. ~P x`),
     PROOF
     (CONSIDER [`a:A`] [(1,`~(P:A->bool) a`)] (BY [0][]) THEN
      TAKE [`a:A`] THEN
      ASSUME_A(2,`(P:A->bool) a`) THEN
      A(3,`F`) (BY [1;2][]) THEN
      THUS_A(4,`!y:A. P y`) (BY [3][]))
    ;(0,`!x:A. P x`),
     PROOF
     (CONSIDER [`a:A`] [] (BY [][]) THEN
      TAKE [`a:A`] THEN
      THUS_A(1,`P a ==> !y:A. P y`) (BY [0][]))])));;
```

7.5 Terms in context

The HOL system parses a `term` in an empty context because the HOL implementation is functional. So if we write an expression of type `term` it doesn't have access to the goal state. This means that `` `n` `` will always be read as a polymorphic variable, whatever is in the current goal. If a goal talks about a natural number `` `n` `` of type `` `:num` ``, then to instantiate an existential quantifier with this `` `n` `` one has to write `` EXISTS_TAC `n:num` `` instead of `` EXISTS_TAC `n` ``. Generally this doesn't get too bad but it is irritating.

In our Mizar mode this problem is worse because there are more statements in the tactics. So we might try to modify things for this. The idea is to change the type `term` to `goal` $\rightarrow$ `term` everywhere. This means that the 'term parsing function' `X` will have to get the type `string` $\rightarrow$ `goal` $\rightarrow$ `term`. Again a variant function `prove''` of type `(goal` $\rightarrow$ `term)` $\times$ `tactic` $\rightarrow$ `thm` is needed.

If we follow this approach then most type annotations will be gone, except in the statement of the theorem to be proved and in the arguments of `LET` and `CONSIDER` (where they also are required in the 'real' Mizar).

Because in that case the terms are parsed in the context of a goal, we can give a special meaning to the variables `` `antecedent` `` and `` `thesis` ``. See Section 8 for an example of this.

The main disadvantage of this modification to our Mizar mode is that the original HOL proof scripts will not work anymore because the `X` function has been changed. That is a big disadvantage if one wants true integration between the 'pure' HOL and the Mizar mode.

7.6 Out of sequence labels and negative labels

Another thing that can be changed is to be less restrictive which numbers are allowed for the labels. Until now they had to be the exact position that the statement would end up in the assumption list of the goal. However there is no reason not to allow any number and put the statement at that position, padding the assumption list with `` `T` `` `thms` if necessary. That way we can make the labels in the example match the labels in the original Mizar version. If we do this in the example proof, then the second `CONSIDER` will see a goal like:

```
it : goalstack = 1 subgoal (1 total)

 0 [`T`]
 1 [`T`]
 2 [`T`]
 3 [`T`]
 4 [`T`]
 5 [`!x. P x`]

`?x. P x ==> (!y. P y)`
```

Related to this is an enhancement that implements Mizar's `then`. In Mizar a step can refer to the step directly before it with the `then` prefix. A way to imitate this in our Mizar mode is to allow negative numbers in the labels, counting back from the top of the assumption stack. The label `-1` will then refer to the top of the stack (which contains the statement from the previous step). Therefore use of `-1` will behave like `then`.

7.7 Symbolic labels

Instead of numeric labels we also can have symbolic labels. HOL Light supports symbolic labels already. It is straight-forward to change the set of Mizar tactics to work with these symbolic labels instead of with the numeric positions in the list of assumptions.

In the rest of this paper we have presented our Mizar mode with numeric labels. We had two reasons for this:

- In HOL the symbolic labels are almost never used, so proof states that contain them are un-HOL-like.
- In Mizar the 'symbolic' labels generally just are `A1`, `A2`, `A3`, ... That means that the Mizar labels really are used as numbers, most of the time. Therefore we didn't consider numeric labels un-Mizar-like.

7.8 Error recovery

A declarative proof contains explicit statements for all reasoning steps. Because of this a declarative system like Mizar can recover from errors and continue checking proofs after the first error.[3] This behavior can be added to our Mizar mode for HOL by catching the exception if there is an error, and then continue with the appropriate statement added to the context as an axiom. This was implemented by using a justification function that just throws an exception.

Having this enhancement of course only gives error recovery for 'reasoning errors' and will not help with other errors like syntax errors or ML type errors.

One of the referees of this paper liked the idea of error recovery for the Mizar tactics, and suggested a stack on which the goalstates of the erroneous steps could be stored for later proof debugging. We implemented this idea, but we think that the standard HOL practice of running a proof one tactic at a time is more convenient (for which the 'Mizar tactics with error recovery' are unusable).

8 Bigger example

In this section we show a larger example of our Mizar mode (it uses the variant from Section 7.5, so terms are parsed in the context of the proof):

```
let FIXPOINT = prove''
 (`!f. (!x:A y. x <= y /\ y <= x ==> (x = y)) /\
       (!x y z. x <= y /\ y <= z ==> x <= z) /\
       (!x y. x <= y ==> f x <= f y) /\
       (!X. ?s. (!x. x IN X ==> s <= x) /\
                (!s'. (!x. x IN X ==> s' <= x) ==> s' <= s))
       ==> ?x. f x = x`,
```

[3] The Mizar mode by John Harrison does not have this feature. Isar satisfies the principle that sub-proofs can be checked independently, but the present implementation simply stops at the first error it encounters.

```
LET [`f:A->A`] THEN
ASSUME_A(0,`antecedent`) THEN
A(1,`!x y. x <= y /\ y <= x ==> (x = y)`) (BY [0][]) THEN
A(2,`!x y z. x <= y /\ y <= z ==> x <= z`) (BY [0][]) THEN
A(3,`!x y. x <= y ==> f x <= f y`)(BY [0][]) THEN
A(4,`!X. ?s. (!x. x IN X ==> s <= x) /\
             (!s'. (!x. x IN X ==> s' <= x) ==> s' <= s)`)
  (BY [0][]) THEN
CONSIDER [`Y:A->bool`] [(5,`Y = {b | f b <= b}`)] (BY [][]) THEN
A(6,`!b. b IN Y = f b <= b`) (BY [5][IN_ELIM_THM;BETA_THM]) THEN
CONSIDER [`a:A`] [(7,`!x. x IN Y ==> a <= x`);
  (8,`!a'. (!x. x IN Y ==> a' <= x) ==> a' <= a`)] (BY [4][]) THEN
TAKE [`a`] THEN
A(9,`!b. b IN Y ==> f a <= b`)
 (LET [`b:A`] THEN
  ASSUME_A(9,`b IN Y`) THEN
  A(10,`f b <= b`) (BY [6;9][]) THEN
  A(11,`a <= b`) (BY [7;9][]) THEN
  A(12,`f a <= f b`) (BY [3;11][]) THEN
  THUS_A(13,`f a <= b`) (BY [2;10;12][])) THEN
A(10,`f(a) <= a`) (BY [8;9][]) THEN
A(11,`f(f(a)) <= f(a)`) (BY [3;10][]) THEN
A(12,`f(a) IN Y`) (BY [6;11][]) THEN
A(13,`a <= f(a)`) (BY [7;12][]) THEN
THUS_A(14,`thesis`) (BY [1;10;13][]));;
```

This is a translation to our framework of an example proof from John Harrison's Mizar mode which proves the Knaster-Tarski fixpoint theorem.

9 Conclusion

The tactics that are presented here might be the basis of a realistic system that offers the best of both the procedural and declarative provers. One hopes this to be possible: to build a prover that has the readable proofs of the declarative provers and the programmability of the procedural ones. The Mizar mode for HOL by John Harrison and the Isar mode for Isabelle might claim to be just that, but in those systems it feels like one has to learn *two* provers if one wants to be able to use both styles of proving.

The Mizar mode of this paper makes clear that that both kinds of prover are very similar. Although the proofs using our Mizar tactics look awkward and fragile compared with their Mizar counterparts, we have shown that it is possible to bridge the gap between the procedural and declarative proof styles in a more intimate way than had been accomplished thus far.

Acknowledgements. Thanks to Dan Synek, Jan Zwanenburg and the anonymous referees for valuable comments. Due to space limits we have not been able to incorporate all of them.

References

1. Henk Barendregt. The impact of the lambda calculus. *Bulletin of Symbolic Logic*, 3(2), 1997.
2. Bruno Barras, e.a. *The Coq Proof Assistant Reference Manual*, 2000. `<ftp://ftp.inria.fr/INRIA/coq/V6.3.1/doc/Reference-Manual-all.ps.gz>`.
3. Gertrud Bauer. Lesbare formale Beweise in Isabelle/Isar — der Satz von Hahn-Banach. Master's thesis, TU München, November 1999. `<http://www.in.tum.de/~bauerg/HahnBanach-DA.pdf>`.
4. Gertrud Bauer. The Hahn-Banach Theorem for real vector spaces. Part of the Isabelle99-2 distribution, `<http://isabelle.in.tum.de/library/HOL/HOL-Real/HahnBanach/document.pdf>`, February 2001.
5. Gertrud Bauer and Markus Wenzel. Computer-assisted mathematics at work — the Hahn-Banach theorem in Isabelle/Isar. In Thierry Coquand, Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs: TYPES'99*, volume 1956 of *LNCS*, 2000.
6. Jan Cederquist, Thierry Coquand, and Sara Negri. The Hahn-Banach Theorem in Type Theory. In G. Sambin and J. Smith, editors, *Twenty-Five years of Constructive Type Theory*, Oxford, 1998. Oxford University Press.
7. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993.
8. M.J.C. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer Verlag, Berlin, Heidelberg, New York, 1979.
9. John Harrison. A Mizar Mode for HOL. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics, TPHOLs '96*, volume 1125 of *LNCS*, pages 203–220. Springer, 1996.
10. John Harrison. *The HOL Light manual (1.1)*, 2000. `<http://www.cl.cam.ac.uk/users/jrh/hol-light/manual-1.1.ps.gz>`.
11. David A. McAllester. *Ontic: A Knowledge Representation System for Mathematics*. The MIT Press Series in Artificial Intelligence. MIT Press, 1989.
12. M. Muzalewski. *An Outline of PC Mizar*. Fondation Philippe le Hodey, Brussels, 1993. `<http://www.cs.kun.nl/~freek/mizar/mizarmanual.ps.gz>`.
13. R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, Amsterdam, 1994.
14. L.C. Paulson. *The Isabelle Reference Manual*, 2000. `<http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/Isabelle99-1/doc/ref.pdf>`.
15. Don Syme. Three Tactic Theorem Proving. In *Theorem Proving in Higher Order Logics, TPHOLs '99*, volume 1690 of *LNCS*, pages 203–220. Springer, 1999.
16. M. Wenzel. *The Isabelle/Isar Reference Manual*. TU München, München, 1999. `<http://isabelle.in.tum.de/doc/isar-ref.pdf>`.
17. F. Wiedijk. Mizar: An impression. `<http://www.cs.kun.nl/~freek/mizar/mizarintro.ps.gz>`, 1999.
18. Vincent Zammit. On the Implementation of an Extensible Declarative Proof Language. In *Theorem Proving in Higher Order Logics, TPHOLs '99*, volume 1690 of *LNCS*, pages 185–202. Springer, 1999.

A Implementation

Here is the full listing of the Mizar implementation as described in Section 6.

```
let miz_error msg nl =
  failwith (rev_itlist (fun s t -> t^" "^s) (map string_of_int nl) msg);;

let MIZ_ERROR_TAC msg nl = fun g -> miz_error msg nl;;

let N_ASSUME_TAC n th (asl,_ as g) =
  if length asl = n then ASSUME_TAC th g else miz_error "N_ASSUME_TAC" [n];;

let A (n,tm) tac =
  SUBGOAL_THEN tm (N_ASSUME_TAC n) THENL
  [tac THEN MIZ_ERROR_TAC "A" [n]; ALL_TAC];;

let ASSUME_A (n,tm) =
  DISCH_THEN (fun th -> if concl th = tm then N_ASSUME_TAC n th
    else miz_error "ASSUME_A" [n]);;

let (BECAUSE:(thm list -> tactic) -> int list -> thm list -> tactic) =
  fun tac nl thl (asl,_ as g) ->
    try tac ((map (fun n -> snd (el (length asl - n - 1) asl)) nl) @ thl) g
    with _ -> ALL_TAC g;;

let BY = BECAUSE (fun thl -> REWRITE_TAC thl THEN MESON_TAC thl);;

let CONSIDER =
  let T = `T` in
  fun vl ntml tac ->
    let ex = itlist (curry mk_exists) vl
      (itlist (curry mk_conj) (map snd ntml) T) in
    SUBGOAL_THEN ex
      ((EVERY_TCL (map X_CHOOSE_THEN vl) THEN_TCL EVERY_TCL (map
         (fun (n,_) tcl cj ->
           let th,cj' = CONJ_PAIR cj in N_ASSUME_TAC n th THEN tcl cj')
       ntml)) (K ALL_TAC)) THENL
    [tac THEN MIZ_ERROR_TAC "CONSIDER" (map fst ntml); ALL_TAC];;

let LET = MAP_EVERY X_GEN_TAC;;

let PER_CASES =
  let F = `F` in
  fun tac cases ->
    let dj = itlist (curry mk_disj) (map (snd o fst) cases) F in
    SUBGOAL_THEN dj
      (EVERY_TCL (map (fun case -> let n,_ = fst case in
        (DISJ_CASES_THEN2 (N_ASSUME_TAC n))) cases) CONTR_TAC) THENL
    ([tac] @ map snd cases) THEN MIZ_ERROR_TAC "PER_CASES" [];;

let TAKE = MAP_EVERY EXISTS_TAC;;

let THUS_A (n,tm) tac =
  SUBGOAL_THEN tm ASSUME_TAC THENL
  [tac THEN MIZ_ERROR_TAC "THUS_A" [n]
  ;POP_ASSUM (fun th -> N_ASSUME_TAC n th THEN REWRITE_TAC[EQT_INTRO th])];;
```