Fig. 4. Pretty Printing Proof Space

```
#PROOF_SPACE();;

asm0 : (* Assumption *)
   . |- p |--> q
asm1 : (* Assumption *)
   . |- b |--> c /\ c |--> q
asm2 : (* Assumption *)
   . |- a unless b
lem10 : (* HINT: asm1, |--> is transitive *)
   ... |- b |--> q
lem20 : (* HINT: asm0, asm2, PSP Law *)
   .. |- (p AND a) |--> ((q AND a) OR b)
lem30 : (* HINT: lem10, lem20, Cancellation Law *)
   ... |- (p AND a) |--> ((q AND a) OR q)
lem40 : (* HINT: Lem30, |--> is monotonic *)
   ... |- (p AND a) |--> q
() : void
```

think machine-like to increase his productivity. Likewise, to increase convenience the machine should be equipped with facilities to support human-level reasoning. Then one will at least have the choice whether to be productive or to work conveniently. Moreover this will make mechanical proving more accessible for the uninitiated. A good style of mechanical proving is, in my opinion, to be found in the border between productivity and convenience.

5 Acknowledgement

I wish to thank Doaitse Swierstra for his patience and resourcefulness in improving this paper. I also thank Carl Seger for encouraging me to start writing this paper.

References

1. Cambridge University. *The HOL System Tutorial*, 1991. source included in the standard HOL package.
2. K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation*. Addison-Wesley Publishing Company, Inc., 1988.
3. J.T. Jeuring. *Theories for Algorithm Calculation*. PhD thesis, Utrecht University, 1993.

This article was processed using the LaTeX macro package with LLNCS style

Fig. 3. Example of LEMMA style

```
LEMMA (`lem30`,"((p:^Pred) AND a) |--> (q AND a) OR q",
       `HINT: lem10, lem20, Cancellation Law`,
   ONCE_REWRITE_TAC [pOR_SYM]
   THEN ASSUME_TAC (RECALL `lem10`)
   THEN ASSUME_TAC (ONCE_REWRITE_RULE [pOR_SYM] (RECALL `lem20`))
   THEN IMP_RES_TAC CANCELATION_LAW) ;;

LEMMA (`lem40`,"((p:^Pred) AND a) |--> q",
       `HINT: Lem30, |--> is monotonic`,
   SUBGOAL_THEN "|== (((q AND a) OR q) IMP (q:^Pred))"
       (\thm. ASSUME_TAC thm)
   THENL [pred_JADE_TAC 3; ALL_TAC]
   THEN ASSUME_TAC (RECALL `lem30`)
   THEN IMP_RES_TAC LT_MONOTONIC) ;;
```

For example if after feeding HOL with the complete proof of the Jade Property, `PROOF_SPACE` will show as in Figure 4. Compare the pretty print with the paper proof given earlier.

3.3 Interactive Mode

The last argument in the `LEMMA` statement is a tactic that actually proves the lemma. The question is of course how one can construct this tactic. The answer again is: the usual way, by working hard in interactive mode. There are only three interactive Proof Space operations: `Lg` (Lemma set Goal) to set a new lemma to be proven, `Le` (Lemma Extend) to replace the standard goal expander `e`, and `LU` (Lemma Undo) to delete a lemma from the Proof Space. Their use is analogous to that of `Dg`, `De`, and `DU` of the DERIVATION package.

4 Conclusion

What the DERIVATION and LEMMA packages offer is, first of all, a convenient way to write your proof as you would do it on paper. This will spare you from having to fuss about machine restrictions at a too early stage. Secondly, it improves proof readability as the proof is now formatted very much in the same way as paper-and-pencil proofs. Thirdly, it keeps record on the expressions or lemmas you have proven so far. You can at any moment request a pretty print of your proof —complete with hints— which looks as close as typewriter font can be to a nicely printed proof on paper.

Finally, I wish to say that it has to be admitted that in mechanical theorem proving —as in any conjunction between man and machine— man should learn to

Prove p AND $a \mapsto q$. To do so we derive:

lem10: { asm1, Transitivity of $\mapsto$ } $b \mapsto q$
lem20: { asm0, asm2, PSP Law } p AND $a \mapsto (q$ AND $a)$ OR b
lem30: { lem10, lem20, Cancellation Law } p AND $a \mapsto (q$ AND $a)$ OR q
lem40: { lem30, Monotonicity of $\mapsto$ } p AND $a \mapsto q$
□

Notice the non-linearity in the proof above: lem30 is inferred from lem10 *and* lem20.

3.1 Setting Up Assumptions

The first thing to do before actually starting the proof is to tell HOL the assumptions the proof may use. This is accomplished by function `Set_Assumptions`, as has already been explained in Section 2. We only want to add that the assumptions are the first lemmas you get for free, and so they are included in the Proof Space. The following will set up the assumptions to prove the Jade Property:

```
#Set_Assumptions
 [(`asm0`,"(p:^Pred)|-->q") ;
  (`asm1`,"(b:^Pred)|-->c /\ c|-->q") ;
  (`asm2`,"(a:^Pred) unless b") ] ;;
```

3.2 Extending Proof Space

To extend the Proof Space with a new lemma we have to tell HOL, using the function `LEMMA`, (a) the new lemma, (b) how to prove it, (c) its name for future references, and (d) a hint as how the lemma was inferred.

```
3.1 : LEMMA : (string # term # string # tactic) -> void
```

`LEMMA (name,lem,hint,tac)` if successful —that is, if `tac` can prove `?- lem`— then it will add `|- lem` to the Proof Space and associate the name `name` to it. `hint` is a string that hints how `|- lem` is inferred.

To access the lemmas already in the Proof Space, `RECALL` is used, which has already been explained in Section 2. Figure 3 shows part of the proof of the Jade Property (lem30 and lem40) in the LEMMA-style. Compare the format with the paper proof given earlier.

One can at any time request a pretty print of the Proof Space, complete with the hints with the function `PROOF_SPACE`.

```
3.2 : PROOF_SPACE : void -> void
```

```
Set_Assumptions [ (‘Asm1‘,A1) ; (‘Asm2‘,A2) ] ;;
LEMMA (‘lem10‘,E1,‘HINT: Asm1, Asm2, PSP Law‘, Tac1) ;;
LEMMA (‘lem20‘,E2,‘HINT: Asm1, Monotonicity of |-->‘, Tac2) ;;
...
LEMMA (‘lem100‘,E,‘HINT: lem10, lem90, Cancel Law‘, Tac10) ;;
```

Do not bother with details yet.The first line setup the assumptions of the proof. Each `LEMMA` statement proves a lemma and adds it to the Proof Space. For example the second `LEMMA` derives `E2` and `Tac2` is the tactic to prove this. The second argument of `LEMMA` is a string that serves as a hint as how the lemma is inferred.

The only primitive operations are: initialization of, addition of a lemma to, deleting a lemma from, and pretty printing the Proof Space. These will be detailed in the next subsections. The package again supports an interactive mode.

The best way to explain how to use the package is again by an example. We will consider an example from programming logic [2]. Consider a non terminating program. For predicates p and q, p unless q means that once p holds it will hold at least until q becomes true, and $p \mapsto q$ means that once p holds then eventually q will hold. The inference rules below hold for these operators. What they operationally mean is not important for our purpose. Our interest is only in the calculation.

Monotonicity of $\mapsto$: $\dfrac{p \mapsto q \ , \ [q \ \mathsf{IMP} \ r]}{p \mapsto r}$

Transitivity of $\mapsto$: $\dfrac{p \mapsto q \ , \ q \mapsto r}{p \mapsto r}$

PSP Law : $\dfrac{p \mapsto q \ , \ r \ \mathsf{unless} \ b}{p \ \mathsf{AND} \ r \mapsto (q \ \mathsf{AND} \ r) \ \mathsf{OR} \ b}$

Cancellation Law : $\dfrac{p \mapsto q \ \mathsf{OR} \ r \ , \ q \mapsto b}{p \mapsto b \ \mathsf{OR} \ r}$

We are interested in proving the following property, which for convenience we will refer as the *Jade Property*.

Jade Property:

$$\forall p, q, a, b, c \bullet (p \mapsto q) \wedge (b \mapsto c) \wedge (c \mapsto q) \wedge (a \ \mathsf{unless} \ b) \ \Rightarrow \ (p \ \mathsf{AND} \ a \mapsto q)$$

A paper proof of the Jade Property goes as follows.

Assume:

asm0	asm1	asm2
$p \mapsto q$	$(b \mapsto c) \wedge (c \mapsto q)$	a unless b

$\nu.(\nu.x) \ll y$. A simple rewriting will prove this. The resulting derivation is shown automatically. Notice that a hint can —as in this case— provide more information than the tactic that proves a derivation step.

```
#Dg("=:bool->bool->bool",
    "(NU (NU x)) << (y:*dom)",
    `HINT: NU is an involution`) ;;

"x << y = (NU(NU x)) << y"
[ "!x. NU(NU x) = x" ]
[ "!x y. x PLUS y = y PLUS x" ]
[ "!x y. d << (x PLUS y) = (NU x) << y" ]
() : void

#De(ASM_REWRITE_TAC[]) ;;
OK..
goal proved
. |- x << y = (NU(NU x)) << y

 "Derivation" (*  *)
    "x << y"
 "$=" (*HINT: by reflexivity of NU *)
    "(NU(NU x)) << y"
() : void
```

To undo the last derived expression, use `DU`. *It is however not a replacement of* `b`*, the backup function, of the subgoal package.*

```
2.9 : DU : void -> void
```

3 The LEMMA Package

As said before in Section 1, a proof is most naturally a set structure and as such we want to to represent proofs in HOL. A *Proof Space* is essentially a set of the lemmas proven so far. To access the lemmas it is necessary that a unique name is associated with each lemma. It is up to the user to choose the names. When proving a new lemma, one is free to use any lemmas existing in the Proof Space. Assumptions can be considered as initialization of the Proof Space.

The package uses global variables too, so nesting is again not possible. This has been done so accessing lemmas becomes simpler. Indeed, we trade flexibility for simplicity and hope that in practice people will benefit more from the latter.

Let us assume we want to prove `|- E`, given `A1` and `A2` as assumptions. A proof using the LEMMA Package looks essentially as follows:

```
#DERIVATION();;

"Derivation" (* *)
    "x << y"
"$=" (*HINT: NU is an involution*)
    "(NU(NU x)) << y"
"$=" (*HINT: by Unit Design Equation*)
    "d << ((NU x) PLUS y)"
"$=" (*HINT: PLUS is commutative and Unit Design Equation*)
    "(NU y) << (NU x)"

() : void
```

2.4 Interactive Derivation

The last argument in the `DERIVE` statement is a tactic that justifies the derivation step. The question is of course how one can construct this tactic. The answer is: in the usual way, by working interactively. I assume the reader is familiar with this practice; otherwise he might consult the HOL Tutorial book [1].

There are only three operations for interactive derivation : `Dg` (Derivation step set Goal) to set the target expression of a derivation step, `De` (Derivation Extend) to replace the standard goal expander `e`, and `DU` (Derivation Undo) to undo a derivation step.

2.7 `: Dg : (term # term # string) -> void`

Let `X` be the last derived expression so far, `<<` be the base relation, and `Asml` be the list of assumptions of the derivation. `Dg (R,Y,Hint)` invokes the subgoal package and makes `?- R X Y` as the goal and `Asml` as the assumptions. For the derivation to make sense `R` must be either `<<` or `=`. `Hint` is a string representing the hint associated with the derivation step.

One is free to manipulate the goal thus created by using `b`, `e` and `r` as usual. One can however opt to use `De` instead of `e` to automatically extend the derivation.

2.8 `: De : tactic -> void`

`De` works in the same as `e`, except that when the goal is proven, the derivation is automatically extended with the expression which was mentioned in the call to `Dg`. Also the resulting derivation is then printed.

The figure below shows how to use `Dg` to setup the derivation of $\nu.(\nu.x) \ll y$ from $x \ll y$. Notice that this actually means proving the equality $x \ll y \ =$

make sense `R` should be either `<<` or `=`. `Hint` is a string representing the hint associated with the derivation of `Y` from `X`. In case of a successful `DERIVE`, the theorem `|- R X Y` is also printed.

The complete HOL proof of ν inversion using `DERIVE` is displayed below. It definitely looks neater then the proof with `SUBGOAL_THEN` as displayed in the earlier section.

```
DERIVE("=:bool->bool->bool",
       "(NU (NU x)) << (y:*dom)",
       `HINT: NU is an involution`,
       REWRITE_TAC[RECALL `NU_INVO`]) ;;

DERIVE("=:bool->bool->bool",
       "d << (NU x) PLUS (y:*dom)",
       `HINT: by Unit Design Equation`,
       REWRITE_TAC [RECALL `UNIT_DE`]) ;;

DERIVE("=:bool->bool->bool",
       "NU y << (NU (x:*dom))",
       `HINT: PLUS is commutative and Unit Design Equation`,
       ONCE_REWRITE_TAC [RECALL `PLUS_COM`]
       THEN REWRITE_TAC[RECALL `UNIT_DE`]) ;;
```

Let X be the starting expression, Y the last derived expression, and $\ll$ the base relation. Thus Y is derivable from X. Assuming that each derivation step consistently preserves the $\ll$ relation, it follows from the transitivity of $\ll$ that $X \ll Y$ holds. To extract this theorem from the derivation use the function `ETD` (Extract Theorem from Derivation).

2.5 `: ETD : void -> thm`

For example, after the complete derivation of ν inversion is fed to HOL, `ETD` will return the ν inversion theorem:

```
#ETD();;
... |- x << y = (NU y) << (NU x)
```

At any moment one can request a pretty print of the derivation constructed so far with the function `DERIVATION`.

2.6 `: DERIVATION : void -> void`

For example after the complete proof of ν inversion is fed to HOL, `DERIVATION` will print as displayed below. Notice how much it looks like the derivation in Figure 1.

label is used to refer to the assumption. The following sets up the assumptions of our example derivation:

```
#Set_Assumptions
   [(`UNIT_DE`, "!(x:*dom) y. d << (x PLUS y) = (NU x) << y") ;
    (`PLUS_COM`,"!(x:*dom) y. x PLUS y = y PLUS x") ;
    (`NU_INVO`, "!x:*dom. NU(NU x) = x") ] ;;
```

An assumption (in theorem form) can be accessed using the function `RECALL`.

```
2.2 : RECALL : string -> thm
```

The argument is a string which is the label of the assumption we wish to access. This should be the same label set up by function `Set_Assumptions`. For example `RECALL `NU_INVO`` will return `. |- !x. NU(NU x) = x`.

Having set the assumptions, to start a derivation one has to tell HOL both the base transitive relation and the starting expression. This is done with the function `BD` ('Begin Derivation').

```
2.3 : BD : term -> term -> void
```

The function has two arguments. The first is the preorder relation upon which the derivation should be based. The second is the starting expression. This function initializes various global variables, and as a side effect it redisplays the list of assumptions, the base relation, and the begin expression. *The base relation has to be transitive* for the derivation to make sense. The following starts the derivation of ν inversion:

```
BD "=:bool->bool->bool" "(x:*dom) << y" ;;
```

2.3 Extending the Derivation

To extend the derivation with a new expression one has to provide (a) the new expression, (b) the relation between the new expression and the last expression derived so far —this is either the base relation or equality—, (c) the hint, and (d) the tactic that justifies the derivation step. This is done with the function `DERIVE`.

```
2.4 : DERIVE : (term # term # string # tactic) -> void
```

Let say that `X` is the last derived expression and that `<<` is the base preorder of the derivation. Then the call `DERIVE (R,Y,Hint,Tac)`, if it succeeds —that is, if Tac proves `?- R X Y`—, will add `Y` to the derivation. For the derivation to

$A1$ and $A2$. Then a proof with the DERIVATION package looks like:

```
Set_Assumptions [ (`asm1`, A1); (`asm2`, A2) ] ;;
BD "<<" E ;;
DERIVE ("<<", E1, `HINT: + is monotonic`, Tac1) ;;
DERIVE ("<<", E2, `HINT: asm2 and e is unit of +`, Tac2) ;;
...
DERIVE ("<<", G,`HINT: follows from asm1`, TacLast) ;;
```

Do not bother with the precise details yet. The first line sets up the assumptions of the derivation. The second line indicates the base relation of the derivation and the starting expression. As we are interested in proving $E \ll G$ then $\ll$ is the obvious choice of the base relation. The starting expression is of course E.

Each `DERIVE` statement is a derivation step. For example the second `DERIVE` derives $E2$ from $E1$ and `Tac2` is the tactic which does the work —more precisely the tactic should prove `?- E1 << E2`. The third argument of `DERIVE` is just the hint associated with the derivation step.

Use `DERIVE` *as if you prove your theorem on paper. The hint part of* `DERIVE` *contains a human-readable information that motivates the justification of the derivation step. The tactic part is the machine codes detailing the precise rewriting and resolution that justify the step.*

One can also at any time pretty-print the derivation so far. The format is very much like in Figure 1.

In further subsections we will discuss each function in further detail. How to use the DERIVATION package is best explained by an example. We will again prove the ν inversion theorem. We consider a general triple $(d, +, \nu)$. Under the assumptions that:

1. ν is an involution, that is, $\forall x \bullet \nu.(\nu.x) = x$
2. $+$ is commutative
3. Unit Design Equation is satisfied, that is, $\forall x, y \bullet d \ll x + y \ = \ \nu.x \ll y$

we will prove the ν inversion theorem, that is: $x \ll y \ = \ \nu.y \ll \nu.x$.

2.2 Setting up the Derivation

The first thing to do is to *set up your assumptions*. This is done with function `Set_Assumptions`.

2.1 `: Set_Assumptions : (string # term) list -> void`

The argument is a list of pair (L_i, A_i) where A_i is a boolean term representing i-th assumption and L_i is a string representing the label of assumption A_i. The

the linear proof style as in Figure 1.

2 The DERIVATION Package

The idea is quite similar to the WINDOW library by Jim Grundy. The actual package however is simpler to use and we expect that one will find it easier to learn.

A derivation, viewed as an object, is a list

$$E_1, \ldots, E_i, \ldots$$

such that $E_i = E_{i+1}$. In particular, by transitivity of $=$, $E_1 = E_{last}$ holds, where E_{last} is the last expression in the derivation. *Extending the derivation* with a new expression E basically corresponds to proving $E_{last} = E$.

Such a sequence construction can be generalized to a derivation with respect to some transitive relation R, which is not necessarily an equality. In this case, either $E_i \ R \ E_{i+1}$ or $E_i = E_{i+1}$ should hold. The transitive relation in the derivation in of ν inversion however is just the boolean equality.

The basic idea is to extend HOL with derivation as a new class of objects. The primitive operations are: initialization, extension, undoing, pretty printing, and extracting a theorem. These are the services provided by the package, and are supported both in interactive as well as non-interactive modes.

The mechanism is built on top of the existing proof mechanisms. This means one can use HOL as one is used to, but with the additional possibility to prove a theorem in a derivation-style. The mechanism employs global variables, which prevents the construction of nested derivations. We argue that people should keep each derivation step sufficiently small such that a one or two line hint is enough for others to see why and how the step was taken. If one should need a sub-derivation to justify a derivation step, this is a good indication that the step taken might be to big.

The global variables represent the current derivation and the transitive relation upon which the derivation is based, and furthermore the assumptions of the current derivation.

A good habit is to leave a hint for every derivation step. If the step is trivial, then say so in the hint such that those who later read it will be at least aware of the triviality and can feel safe to skip it. We thus included such hints in the HOL internal representation of a derivation, and consequently the user is required to explicitly supply such hints when extending a derivation.

This is all we say here about the internal mechanism involved, since most users are not required to be aware of it. For our purpose here, *'how to use'* the package is more important than *'how it works'*. People eager enough for the latter will probably find no difficulty in reading the code.

2.1 How a Proof with the DERIVATION Package Looks Like?

Suppose we wish to prove $E \ll G$, where $\ll$ is some transitive relation (examples of such a relation: $=, \Rightarrow, \Leftarrow$, and $<$). Let us assume that we are allowed to assume

some rewritings can be compacter written than others or because some resolution will create fewer garbage assumptions. A good style should not make explicit such mechanical considerations.

One may indeed ask why we bother about the logic of the proof at all. What counts is that the theorem is proven. The *'how it is proven'* is only interesting to display to let other people verify its correctness. Since HOL guarantees correctness, such *display for verification* is no longer necessary. However, verification is not the only purpose of displaying proofs. The proof may give more insight into the nature of a problem than the theorem itself. It may give a good example of an elegant use of the calculus involved. It may also involve some clever trick which may inspire other proofs. Also, one may wish to change or extend the representation of an abstract domain. This implies re-doing the proofs and modifying them accordingly. Unfortunately if the proofs were done quite some time ago —or worse, they were written by someone else— then this philosophy of *'a theorem, once proven, is all one needs to know'* will definitely cause a lot of annoying re-thinking.

There is of course a neater way of giving proofs using HOL. One can use the WINDOW library by Jim Grundy to mimic the proof style in Figure 1. One can also use the `SUBGOAL_THEN` tactical, a style which is displayed below.

```
#let thm = TAC_PROOF
   (([UNIT_DE; PLUS_COM; NU_INV0],
    "!x (y:*dom). x<<y = (NU y) << (NU x)"),
    REPEAT GEN_TAC
    THEN SUBGOAL_THEN
      "(x:*dom) << y  =  NU (NU x) << y"
      (\thm. REWRITE_TAC[thm])
    THENL [ REWRITE_TAC [ASSUME NU_INV0] ; ALL_TAC]
    THEN SUBGOAL_THEN
      "NU (NU (x:*dom)) << y  =  d << (NU x) PLUS y"
      (\thm. REWRITE_TAC[thm])
    THENL [ REWRITE_TAC [ASSUME UNIT_DE] ; ALL_TAC]
    THEN SUBGOAL_THEN
      "(d:*dom) << ((NU x) PLUS y)  =  (NU y) << (NU x)"
      (\thm. REWRITE_TAC[thm])
    THEN  ONCE_REWRITE_TAC [ASSUME PLUS_COM]
      THEN REWRITE_TAC[ASSUME UNIT_DE]) ;;
```

Indeed, the proof is considerably larger than the proof in Figure 2, but do not be discouraged by its size. For a sufficiently complicated theorem this is a more than fair price to pay as the logic and flow of the proof is now made explicit.

Still, this `SUBGOAL_THEN` mechanism can be enhanced further in order to facilitate and encourage even better proof styles, especially by removing the necessity to provide excessive information and by improving the visual appearance. Section 2 discusses the DERIVATION package, an extension to HOL to support

it may not be very precise by only mentioning the most important trick of the particular derivation step. The step is usually small enough for the reader to reconstruct the missing information —usually they are either trivialities or well known facts. For example hint (1) is very precise while hint (3) assumes the reader is aware that + is commutative. Indeed, such a triviality is best verified by a machine.

This style has been successfully applied, especially in the calculational derivation of algorithms from specifications [3]. We ourselves have practiced it with a great pleasure over the years and we think it will be a great enhancement if mechanical proving environments would also support such proofs. This is not to suggest to only use the machine as a piece of paper. We readily admit that paper and machine are two basically different proof mediums. Still, when dealing with abstract domains where finiteness is not a necessary property —or not even desired— mechanical proving becomes an extensive interaction between man and machine. That is why we advocate that a successful, mechanically supported proof style should be a combination of the typical machine proof styles and paper-and-pencil proof styles.

HOL is definitely powerful enough to derive the ν inversion. A typical (non-interactive) proof is given in Figure 2.

Fig. 2. Typical HOL Proof of ν Inversion

```
#let thm = TAC_PROOF
    (([UNIT_DE; PLUS_COM; NU_INVO],
     "!x (y:*dom). x<<y = (NU y) << (NU x)"),
     REWRITE_TAC [SYM(SPEC_ALL (ASSUME UNIT_DE))]
     THEN ONCE_REWRITE_TAC[ASSUME PLUS_COM]
     THEN REWRITE_TAC (map ASSUME [NU_INVO; UNIT_DE])) ;;

thm = ... |- !x y. x << y = (NU y) << (NU x)
```

The proof in Figure 2 is definitely short, but unfortunately it does not tell you very much about *how* the theorem is proved. It does tell you about what facts are used in the proof —for example the commutativity of + is used (`PLUS_COM`) and the Unit Design Equation is used twice (`UNIT_DE`)— but how the derivation is constructed, one has to decipher it from the code, or to find out by running the code in interactive mode. Either way, this is not a convenient way to follow the logic of the proof and the linear proof in Figure 1 —despite its informality at some points— will be significantly more appreciated by most human readers.

There is another significant difference. It may be the case that deriving $\nu.y \ll \nu.x$ from $x \ll y$ is more natural than the other way around. Yet in HOL it can be very tempting to follow a different order —as in the proof in Figure 2— or to introduce intermediate expressions which are less natural because, for example,

more conventional mathematicians may cause the remark that they have written PASCAL code better than that, and one will not be able to construct even a clue about the essential proof steps taken.

Without the help of machines people only use resolution and rewriting when it becomes a triviality, or common knowledge, or something which can be convincingly motivated despite the informality. People profit most from their intelligence by systematically breaking up the proof into smaller lemmas. A proof proceeds by proving one lemma after another until one finally comes to the desired one. As the inference of a lemma may depend on any —or even all— previously proven lemmas, a proof is most naturally set structured.

On the other hand, HOL with its Subgoal Package seems to enforce that a proof has a tree-like structure, and if not it has to be molded into such a structure by duplicating parts of the proof. But people (and computer scientists especially) do not like duplication, and so it becomes a sport as how to construct tree-like proofs without duplicating. We do not consider this as a preferable route. A proof should be structured as it seems most natural to one's mind. To influence this naturality with the machine's restrictions is to force man to think machine-like.

Since the lemmas reflect how one's mind views and decomposes the problem and since this is something a machine cannot do, it is best to keep it this way. The machine should however provide an excellent environment for the human to decompose the problem in any desirable way. The machine can contribute at any time whenever a lemma is sufficiently simple to be proven more or less automatically. Section 3 discusses the LEMMA package, an extension to HOL to support set structured proofs.

A special class of proofs is formed by the linear proofs. In a linear proof the derivation of the next expression depends solely on the last derived expression. Linear proofs read nice and are preferred by many. Quite a large number of useful theorems can be proven with linear proofs. Figure 1 displays an example of this style. It shows how $\nu.y \ll \nu.x$ can be derived from $x \ll y$ assuming the *Unit Design Equation*.

Fig. 1. Example of Derivation: ν Inversion

$$
\begin{array}{ll}
 & x \ll y \\
= & \quad \{\ (1)\ \nu \text{ is an involution, that is: } \forall x \bullet\ \nu.(\nu.x) = x\ \} \\
 & \nu.(\nu.x) \ll y \\
= & \quad \{\ (2)\ \text{Unit Design Equation: } \forall x, y \bullet\ d \ll x + y\ =\ \nu.x \ll y\ \} \\
 & d \ll \nu.x + y \\
= & \quad \{\ (3)\ \text{Unit Design Equation}\ \} \\
 & \nu.y \ll \nu.x
\end{array}
$$

The hints tell us how to derive each expression from the previous one although

On the Style of Mechanical Proving

ISWB Prasetya

Rijksuniversiteit Utrecht, Vakgroep Informatica
Postbus 80.089, 3508 TB Utrecht, Nederland
Email: wishnu@cs.ruu.nl

Abstract. It is quite surprising that —after 5 years of experience— there has been still very little written about how to present and construct mechanical proofs. Such guidelines would definitely help a novice. Since mechanical proving is very different from proving on paper, it may take a long time before one gets accustomed to it and develops an efficient style for constructing and presenting such proofs. Traditional styles, like e.g. decomposing a problem into lemmas and incorporating hints to make the logic of a proof visible, are not straightforwardly taken over to the HOL world. In this paper we present two extensions to HOL, the DERIVATION and LEMMA packages, by which proofs can be written in very much the same format as one would on paper. They are not intended as a replacement of existing HOL mechanisms, but rather as an extension for enabling work in a higher level proof environment.

Category: Research Report

1 Combining the Man and the Machine

To many users of the HOL system, even after gaining extensive experience, the convenience of a pencil-and-paper proof would still be very tempting. Of course such proofs do not have the *consistency preserving* property which HOL provides, but on the other hand one is not bothered by complaints if one deliberately has left out the type of some function, or if one deliberately abuse some notation. Even if one is willing to accept the latter as a fair price to pay for the rigorousness offered by the machine, there is still another wonderful aspect of pencil-and-paper proofs missing: the ability to adhere to a specific proof style, resulting in well documented and easily readable proofs.

Machines are very good in doing brute resolution and rewriting, and this makes them unsurpassed for proving properties of finite domains, with tautology checking for propositional logic being the prime example. However when one wants to use machines to assist reasoning about abstract domains, things will very soon explode; especially as abstract domains are usually —and for a good reason too— infinite. One has to manually direct the machine in selecting what and how to rewrite, and which theorems exactly to use for resolution. When the proof is finally completed, a HOL veteran will not be surprised when overlooking the long and obscure code developed during the process. Showing this result to