Proof Accounts in HOL

Avra Cohn

University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge, CB2 3QG, England.

Abstract:

This paper presents a method for extracting explanations of goal-oriented proofs from the process of generating such proofs in the HOL system. The aim has been to produce natural (if stylized) explanations which are phrased in conventional terms, even where the tactics used in generating the proof are specific to HOL, HOL's implementation, or mechanized theorem proving in general. Internal forms of the explanations are constructed by enriching the ML types that support goal-oriented proof in HOL, so that adequate information can be saved during the generation of a proof to enable explicit, annotated proof trees to be produced. These trees are then rendered in readable form by a suite of printing functions.

Acknowledgements:

This work was supported by the Science and Engineering Research Council, on Grant ... . Thanks to Mike Gordon for his assistance, and to everyone in the Cambridge Hardware Verification Group for their interest and comments.

Contents

Table of Figures

1 Introduction

Proof accounts are intended to explain and document HOL[1] proofs in something approaching conventional or textbook terms. They do this for proofs which are generated 'top down' in HOL through the application of *tactics* to *goals*. Tactic and goals in HOL (as in LCF) are *metalanguage* constructs which are used to generate inferences in an underlying formal *logic*. Thus, a proof in the sense of a proof strategy (a procedure expressed as a structure of metalanguage tactics), when applied successfully to a goal, generates a proof in the sense of a chain of primitive inferences culminating in the desired theorem. Proof accounts explain 'proofs' in the former sense.

Generally, top-down (goal oriented) proofs in HOL can be represented by tree structures of 'proof steps', where each step is a tactic. A tactic can be:

- One of HOL's built-in tactics
- The result of applying a tactic-valued function when applied to arguments of appropriate type
- A combination (such as alternation) of existing tactics
- A tactic implemented directly in the metalanguage by a user[2].

The tactics are composed into a tree structure via the metalanguage combinators `THEN` (for sequencing) or `THENL` (for selective sequencing). Thus, for tactics T_1, T_2, $\cdots$, T_n:

- T_1 `THEN` T_2 is a tactic which, given a goal, first applies T_1 to the goal, then applies T_2 to each resulting subgoal.
- T_1 `THENL` $[T_2; \cdots; T_n]$ is a tactic[3]

  which, to produce its results given a goal, first applies T_i respectively to the i results, for i from 2 to n.

[1] The HOL (higher order logic) system is a system designed by Mike Gordon for helping to automate formal proofs in higher order logic. It is based on Robin Milner's LCF system.

[2] This last possibility, however, is not considered in this paper.

[3] This notation denotes the list of elements shown.

Given an initial goal, each step of a proof results in a set of intermediate subgoals, which, if and when established, are adequate to establish the original goal. That is, each proof step computes the function which will map the established subgoals (i.e. theorems) back to a theorem establishing the original goal, via logical inference. Goals are decomposed successively in this way until they yield axioms or previously proved theorems; then the intermediate functions are applied to construct a chain of theorems culminating in the theorem establishing the initial goal.

Proofs in HOL are typically performed during interactive sessions in which tactics are applied to successive goals, in the context of a HOL theory[4]. During a successful interaction, the user is made aware of intermediate subgoals as they are generated by tactics; and in due course, of the theorem that establishes each subgoal. However, this information is ephemeral, and is available only at certain times during the interactive session. In the end, all that can be preserved of the working session *within* the HOL theory is the final theorem itself. This is adequate in that the type system of HOL's metalanguage assures that no theorem can be computed except by inferences in the logic[5]; and the logic itself has been shown consistent (Pitts, manual ref). However, should a user wish to know more about the way in which a proof was accomplished after the working session is finished, none of the intermediate goals or theorems will have been saved in the relevant HOL theory.

As the HOL system stands, the only persistent record that can be kept of the way in which a formal proof was produced is the text file that a user keeps – optionally, of course – in order to document the interactive session. (Most users do preserve, in some systematic way, the metalanguage procedures that prove their theorems.) Records of this sort are, however, extraneous to the formal logic or any theory extending the logic; they associate only informally with such theories.

In any case, the metalanguage text which generates a proof is not necessarily, in itself, a useful explanation of the proof strategy. Comments added by the user may help, but inserting comments by hand is tedious, difficult to do in adequate detail, and not guaranteed to be accurate. The metalanguage

[4]A HOL theory corresponds to a logical theory in the standard sense of an extension of a logic via well-founded definitions and deduced theorems.

[5]This use of the type discipline of the metalanguage was Milner's key idea in the LCF system. It dispenses with the need to preserve primitive inference sequences, but without loss of security.

text itself may not be accurately saved, or wholly intelligible to a reader in certain situations. This is so particularly

- For longer or more complex structures of tactics
- For theorem-proving based, technical or HOL-specific tactics
- When proof steps are specified as the result of tactic-valued functions applied to appropriate arguments (such specifications may be arbitrarily nested and complex)
- When tactic-valued functions produce tactics which obscure individual proof steps
- For context or implementation dependent tactics (e.g. a tactic which refers to the 'third current assumption')
- For combined tactics (e.g. combined by the operator '`ORELSE`')
- When previously proved lemmas are denoted simply by name, or are computed *in situ*
- When parallel branches of a proof are treated simultaneously by non-branching strategies
- When expert HOL users rely on personal styles of tactical proof not familiar to other users.

In this paper, we propose what we hope is an intelligible, accurate and informative style of documentation of goal oriented proofs, and a method for deriving proof explanations in this style automatically upon the application of tactics to goals. The purpose of these proof accounts is to clarify and document successfully completed HOL proofs in a style free from HOL-related or theorem-prover based terms and concepts; that is, as close in spirit as is possible to textbook style proof presentations without involving natural language expertise.

Possible future applications of proof accounts might include

- Debugging user-designed tactics

- Tools for teaching HOL
- Tools for improving successful proofs.

Further possible applications are discussed in ... .

Although this paper is probably of most interest to HOL users, and does not contain a presentation of the HOL system, we hope that the main ideas will be clear to other interested readers. Documentation of the HOL system may be found in ().

All of the example sessions and remarks pertain to Version 11 of HOL (1990). Minor modifications for Version 12 (1991) are currently in progress.

1.1 The HOL System

1.1.1 The Metalanguage and Logic

LCF-based systems such as HOL are built around (i) a sequent calculus, and (ii) a programming language (ML, for metalanguage) in which objects of the calculus can be represented and computed. In particular, terms and theorems of the logic can be denoted, and proofs can be computed. This is done by representing rules of inference as metalanguage functions which map theorems (sometimes with various parameters) to new theorems; and implementing these functions as ML procedures.

ML's type system plays an essential role in enabling *theorems* to be protected as abstract types. Thus, one may inspect the conclusion or hypotheses of a theorem (i.e. decompose a theorem into its syntactic parts) but may not construct a theorem from its parts; theorems can be produced only by application of functions expressing rules of inference.

In recent years, the language ML has been interfaced to several logics in the hope of assisting in the proof of theorems in these logics. The original logic (PPLAMBDA) of the LCF system was intended for proofs about recursive functions defined in domains, which are useful in algorithm and software verification. In HOL, a version of Church's higher-order predicate calculus (also called HOL) is used. This is intended for proofs about digital systems, and for other areas in which the issues of definedness and termination are less central. The Nuprl system (...) uses the logic ITT (intuitionistic type theory).

For many applications, the full expressiveness of a general-purpose programming language is not necessary; a set of primitive proof-building operations would suffice. One of the capabilities which ML, as a full programming language, provides – for users experimenting with proof methods, proof styles, automation, and so on – is a way to express and test informal proof strategies of their own design. These strategies can be anything from very simple proof techniques (for example: "In order to prove P, assume $\neg P$ and prove falsity") to sophisticated searching heuristics. However, this paper restricts itself to HOL's main built-in tactics.

1.1.2 Goal Oriented Proof

In both simple and complex cases, the LCF-HOL methodology is geared to the natural 'backward' style of proof often used in textbook presentations: proceding from goal to subgoals via strategies, until recognizably trivial subgoals are reached. Each stage of the decomposition is accompanied by a *justification* function in which is embedded the inference pattern enabling the move from established subgoal to established goals. The justification is again a function: it maps the set of theorems purporting to achieve the respective subgoals to the theorem achieving the original goal – by invoking the inference pattern in question. (A theorem is said to achieve a goal if the conclusion of the theorem is the term of the goal, up to alpha-conversion, and the hypotheses of the theorem are a subset of the assumptions of the goal.)

There are therefore two stages in a tactical proof: the search stage, in which successive subgoals are generated until (and if) axioms or previously established subgoals are produced; and the justification stage in which theorems achieving goals are deduced in succession from theorems achieving their subgoals, via formal inference. These are often thought of as reverse processes, the first producing and working down a tree structure of subgoals, and the second working back up to the original goal.

This proof style, of course, is really no more than a convenient way of presenting a proof, and of dressing the 'real' proof, namely, the sequence of theorems culminating in the desired theorem, where each theorem in the sequence is either an axiom or is a consequence of earlier theorems in the sequence. The style conceals from the user the book-keeping process through which the real proof is contructed as the subgoals are decomposed and eventually

achieved. Thus the simple strategy above ("In order to prove P, assume $\neg P$ and prove falsity") is a presentation of the inference rule: "From the theorem asserting falsity, under the assumption that P is false, derive the theorem asserting P"; the strategy packages the inference rule in a convenient way.

The sequences of theorems culminating in a given theorem are not recorded as a result of performing a goal oriented proof; they are simply computations occurring in time. That is, the function representing each inference rule used is applied to arguments, which in turn means that the ML procedure representing that function is executed. Because inferences are represented as functions, the proof (in the sense of the inference sequence) is an ephemeral part of the computation which represents the goal oriented proof effort.

Proof accounts are based on enhancements of the metalanguage types of goals, tactics and justifications which allow sufficient additional information to be recorded for an explanation of the proof to be generated and preserved.

1.1.3 The Subgoal-Theorem Tree

The tree structure of successive subgoals – together with a record of the proof steps leading from goals to subgoals, and the theorems achieving the various goals – is a concept which is always in the background when tactical proofs are performed in HOL. For example, application of the tactic encoding the strategy above ("In order to prove P, assume $\neg P$ and prove falsity") to an appropriate goal would always produce exactly one subgoal, and this would be achieved by one theorem; the usual numerical induction tactic would produce two subgoals (the base and step cases); case analyses would produce at least two, and so on. However, such trees are neither represented explicitly in HOL nor open to exploration[6].

The structure of achieving theorems forms an essential part of the tree. In a successful top-down proof, there is, for each node (i.e. goal) of the tree, starting at the leaves, a theorem achieving that goal. Where one goal diverges (under the application of a tactic) into several subgoals, the several achieving theorems converge (by inference) to produce one theorem. Thus the numerical induction strategy would induce two subgoals when applied to

[6]The subgoal package in Version 11 of HOL, which is an add-on facility, can be used to manage the subgoal-theorem tree *during* a working session; it is based on a stack representation of the tree. Again, however, this stack is not open to exploration by users; nor is it explicit, or preservable

a goal, *and* a justification function. The justification function at that node would accept the two achieving theorems and produce the theorem achieving the original goal.

The whole tree structure representing the proof thus includes the proof steps, the subgoals, and the achieving theorems. Proof accounts are based on an explicit and preservable representation of this structure of goals, proof steps and theorems.

1.2 An Example Textbook Proof

To give an idea of the textbook style to which proof accounts aspire, we give some fragments of a real example. The proof from which these are taken is from "The Higher Arithmetic" by H. Davenport. The proof is of the uniqueness of prime factorization.

Theorem: *Any natural number can be represented in ... only one way as a product of primes.*

Proof: We prove the uniqueness of factorization by induction. This requires us to prove it for any number n, on the assumption that it is already established for all numbers less than n. If n itself is a prime, there is nothing to prove. Suppose, then, that n is composite, and has two different representations as products of primes, say

$$n = p\ q\ r \cdots = p'\ q'\ r' \cdots,$$

where p, q, r, $\cdots$ and p', q', r', $\cdots$ are all primes. The same prime cannot occur in both representations, for if it did we could cancel it and get two different representations of a smaller number, which is contrary to the induction hypothesis.

$$\vdots$$

Now consider the number n - $p\ p'$. This is a natural number less than n, and so can be expressed as a product of primes in one and only one way.

$$\vdots$$

This contradiction proves that n has only one factorization into primes.

This presentation of the proof has the following features:

- The presentation is in sophisticated but still stylized English, using standard phrases such as "This requires us to prove $\cdots$", "Suppose, then, that$\cdots$" and "Now consider $\cdots$".
- It is generally presented in a goal oriented style, and this requires the reader to maintain his location in the implied subgoal tree (and hence to understand the scope of assumptions such as "Suppose, then, that n is composite").
- Within the goal oriented format there are intervals of forward reasoning; for example, "Now consider the number n - p p'. This is a natural number less than n, and so $\cdots$".
- Minor steps are omitted in places; for example, "If n itself is a prime, there is nothing to prove" – there is, of course.
- The presentation is cast in purely problem-related and logical terms – i.e. it refers to numbers and their properties; to patterns of reasoning such as proof by contradiction; and to standing assumptions such as the induction hypothesis – but to nothing more technical in the realm of theorem-proving.

Our aim is to produce proof accounts which have as many of these properties as possible without approaching the natural language issues. That is, we will be satisfied with pre-packaged phrases in a tiny subset of English, as long as the explanations are structured in something approaching the conventional style, and depend on similar concepts.

1.3 Design Decisions

The current prototype accounting facility rests on the following design decisions:

- The facility is not interactive in the first instance; i.e. is not intended to be used whilst developing a proof, but rather to generate explanations of successfully completed proofs.

- The proof explained is the proof in the sense of the strategy rather than the proof in the sense of the inference sequence. A proof step is taken to be a tactic without internal sequencing. These tactics are taken to be the main proof steps.
- There is an explicit data structure to represent subgoal-theorem trees with proof steps. Each account is a presentation of an instance of this data structure.
- The construction of this tree is separated from its presentation. That is, there is an internal representation of the tree, as well as a set of printing functions for producing a readable rendition.
- We attempt to capture the character of the textbook prose but without any natural language capabilities.
- For the present, the basic HOL tactics are re-implemented to produce accounts, and for this purpose are given distinct names.

Improvements and elaborations are discussed in ().

1.4 Related Work

The only other similar explanation facility we know about is the one provided for the Boyer-Moore theorem-prover (). As we understand it, the present facility differs from that one in the following ways:

- The Boyer-Moore facility explains the action of the (automatic) theorem prover as it searches for a proof. Though it searches very efficiently, the explanation is still given in terms of the search rather than of the proof directly. The facility for HOL aims at explaining the proof found rather than the search process.
- The Boyer-Moore system produces explanatory text in real time, as the proof search is in progress. Ours re-runs completed proofs in order to generate explanations.
- The Boyer-Moore facility does not (apparently) construct an explicit internal respresentation of an explanation, but rather, produces fragments of explanation as a side effect of the proof search. We do aim

at constructing an internal representation – which can itself be transformed, printed, etc.

- The Boyer-Moore facility does give attention to the quality of the natural language produces, while ours does not.

2 The Basic Idea

In this chapter we give an example of a successful proof session in HOL and show, for this proof, the style and content of the explanation being proposed. The accounting facility uses HOL's methods of subgoal decomposition and proof assembly to generate a proof account as a side-effect of performing a goal oriented proof. The information preserved makes it possible to identify certain key 'proof events' such as the solution of a subgoal, the splitting of a goal into subgoals, proof by contradiction, assumptions made behinds the scenes, and invalid use of lemmas.

In the following HOL session, a simple theorem is proved: the associativity of addition. (This is actually one of the theorems that is already proved in the theory of arithmetic when HOL is entered.) The proof uses the theorem called `ADD_CLAUSES`:

```
ADD_CLAUSES =
|- (0 + m = m) /\
   (m + 0 = m) /\
   ((SUC m) + n = SUC(m + n)) /\
   (m + (SUC n) = SUC(m + n))
```

This is the HOL session in which the theorem `ADD_ASSOC` is proved[7].

```
#let g = [],"!m n p. m + (n + p) = (m + n) + p";;
g = ([], "!m n p. m + (n + p) = (m + n) + p") : (* list # term)

#let tac = INDUCT_TAC THEN ASM_REWRITE_TAC[ADD_CLAUSES];;
tac = - : tactic

#let gl,p = tac g;;
gl = [] : goal list
p = - : proof

#let ADD_ASSOC = p[];;
ADD_ASSOC = |- !m n p. m + (n + p) = (m + n) + p
```

[7] In the sessions that follow, we use HOL in mode in which hypotheses of theorems are printed in full; the ML top level printing function has been set to print hypotheses of theorems in full.

In this session, a goal, `g`, is first constructed; it consists of the term to be proved (namely, `"!m n p. m + (n + p) = (m + n) + p"`), together with a list (initially empty) of assumptions which may be used subsequently. A tactic (`tac`) is applied to the goal; the tactic is a function. The tactic is formed by sequencing two of HOL's built-in tactics: a tactic `INDUCT_TAC`, which implements the numerical induction strategy, and a tactic of the form `ASM_REWRITE_TAC` l, (where l is a list of theorems), which implements the strategy of rewriting (simplifying) using (i) the theorems in the list l, (ii) any of HOL's built-in basic rewriting theorems, and also (iii) any assumptions of the goal in question[8] (hence the '`ASM_`' – the tactic `REWRITE_TAC` l would not use the assumptions of the goal).

The application of `tac` to `g` yields a list of goals (`gl`), together with a justification function (`p`). The list of goals represent the collection of subgoals which, if all achieved, would suffice to achieve the original goal. The justification function maps the list of theorems (respectively) satisfying the subgoals to a theorem achieving the original goal. The mapping consists of a sequence of inferences leading from the given theorems to the desired theorem. Thus, the interaction consists in two stages: the generation of subgoals until there are no more subgoals; and the construction of the proof through inference, based on the various justification functions.

The theorem produced can be named and preserved for future use as part of the logical theory in which it was established; and the text of the tactic can be saved in a file (outside of the logical theory); but that is all that can be preserved of the proof process and proof session.

The tactic (`tac`) in this case is so simple that at first sight it would seem to point directly to a proof explanation – which might read:

> To prove `"!m n p. m + (n + p) = (m + n) + p"`, do induction on `m`, and then, for all resulting cases, simplify with the fact `ADD_CLAUSES`, with any current assumptions, and with the basic tautologies.

However, the explanation does not follow so obviously from examination of `tac`. First, the fact that the proof is by induction depends on associating the ML function name '`INDUCT_TAC`' with the strategy of mathematical induction. Second, it actually requires some thought to perceive that the

[8] The assumptions are represented as terms t, so for purposes of rewriting they are considered as theorems of the form $t \vdash t$

induction step produces two subgoals even though the goal is solved by a 'linear' sequence of steps. It also takes some thought to realize that an induction assumption applies in the step case, but not in the basis case (and hence that `ASM_REWRITE_TAC` amounts to `REWRITE_TAC` in the basis case). It requires further thought to state the induction hypothesis precisely. Finally, the name '`ADD_CLAUSES`' does not immediately reveal the theorem or definition denoted by that name. If the tactic were more complex, the pattern of reasoning indicated might be even less obvious.

An equivalent tactic could be formed in this case by selective sequencing; this makes the underlying tree of subgoals, and hence the explanation, a little clearer:

```
#let gl,p =
  (INDUCT_TAC
    THENL [ASM_REWRITE_TAC[ADD_CLAUSES];ASM_REWRITE_TAC[ADD_CLAUSES]]) g;;
##gl = [] : goal list
p = - : proof

#let ADD_ASSOC = p[];;
ADD_ASSOC = |- !m n p. m + (n + p) = (m + n) + p
```

but this form is verbose and thus often avoided.

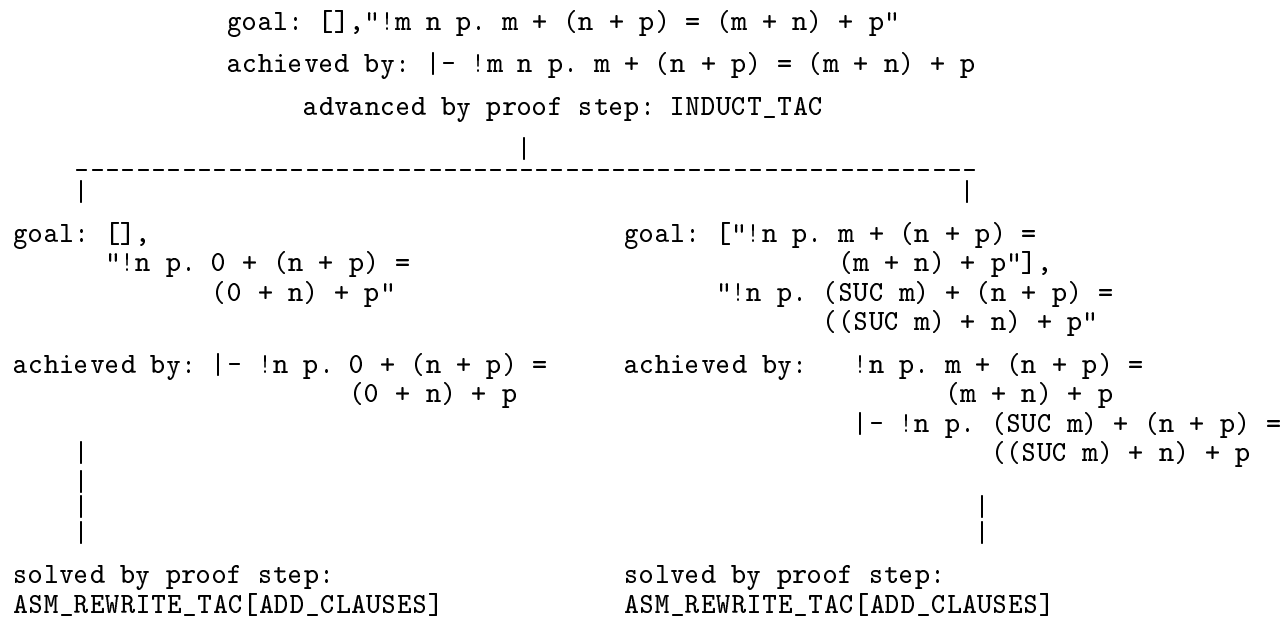
Some further information is revealed by generating the proof in stages. (Normally, the subgoal package would be used to do the book-keeping seen here.) The head and tail of the list (`gl1`) of induction subgoals are computed respectively by the ML functions `hd` and `tl`. Subsequent subgoal lists and justification functions are named as shown:

```
#let gl1,p1 = INDUCT_TAC g;;
gl1 =
[([], "!n p. 0 + (n + p) = (0 + n) + p");
 (["!n p. m + (n + p) = (m + n) + p"],
  "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p")]
: goal list
p1 = - : proof

#let gl2,p2 = ASM_REWRITE_TAC[ADD_CLAUSES](hd gl1);;
gl2 = [] : goal list
p2 = - : proof

#let th2 = p2[];;
th2 = |- !n p. 0 + (n + p) = (0 + n) + p

#let gl3,p3 = ASM_REWRITE_TAC[ADD_CLAUSES](hd(tl gl1));;
gl3 = [] : goal list
p3 = - : proof
```

```
#let th3 = p3[];;
th3 =
!n p. m + (n + p) = (m + n) + p
|- !n p. (SUC m) + (n + p) = ((SUC m) + n) + p

#let ADD_ASSOC = p1[th2;th3];;
ADD_ASSOC = |- !m n p. m + (n + p) = (m + n) + p
```

Here, the list of subgoals, `gl1`, shows explicitly the two intermediate subgoals produced by the induction step, and it can be seen how each is subsequently affected by the rewriting step, and finally achieved by a theorem. However, though they can be viewed, the goals, steps and theorems are neither structured into a tree nor preserved, but are simply bound to ML identifiers for the duration of the particular HOL session in which they occur. The meaning of the name `ADD_CLAUSES` is still not explicit; and the reasoning pattern denoted by '`INDUCT_TAC`' still depends on knowing the names and effects of the built-in ML functions.

The completed tree, if it could be seen now, might look something like this[9]:



[9]How it 'looks' depends on the conventions for displaying it, of course.

Using the subgoal package[10], the subgoal-theorem tree is represented (but only implicitly within HOL) using stacks. However, the tree cannot be searched or examined, except by proceding with (or undoing) the interactive proof, and it cannot be preserved; and the problems of `ADD_CLAUSES` and `INDUCT_TAC` still remain. In the session below, the command `set_goal` has the side effect of putting a goal on the goal stack, and a command of the form `expand` *tac* applies *tac* to the goal at the top of the stack. Sibling subgoals are stacked in left-to-right order, and the subgoal tree is traversed in left-to-right order. A useful reminder of the next remaining subgoal is printed when a goal is achieved. (Note that the hypotheses of a theorem is printed as '.'.)

```
#set_goal([],"!m n p. m + (n + p) = (m + n) + p");;
"!m n p. m + (n + p) = (m + n) + p"

() : void

#expand INDUCT_TAC;;
OK..
2 subgoals
"!n p. (SUC m) + (n + p) = ((SUC m) + n) + p"
    [ "!n p. m + (n + p) = (m + n) + p" ]

"!n p. 0 + (n + p) = (0 + n) + p"

() : void

#expand(ASM_REWRITE_TAC[ADD_CLAUSES]);;
OK..
goal proved
|- !n p. 0 + (n + p) = (0 + n) + p

Previous subproof:
"!n p. (SUC m) + (n + p) = ((SUC m) + n) + p"
    [ "!n p. m + (n + p) = (m + n) + p" ]

() : void

#expand(ASM_REWRITE_TAC[ADD_CLAUSES]);;
OK..
goal proved
. |- !n p. (SUC m) + (n + p) = ((SUC m) + n) + p
|- !m n p. m + (n + p) = (m + n) + p

Previous subproof:
goal proved
() : void
```

The proof account facility produces the following explanation of the same proof. It does so as a result of applying to a goal based on the original goal a

[10] of HOL Version 11 – that of HOL Version 12 is more sophisticated

tactic based on the given tactic. The marker >>>> indicates a proof step, and >>, a goal to be achieved. The subgoal tree is presented depth-first, left to right. Theorems are shown as they are achieved. Each return to a pending subgoal is remarked:

```
This is the proof of the conjecture
>> ADD_ASSOC:
   "!m n p. m + (n + p) = (m + n) + p"

>>>> The proof is by mathematical induction on "m".
     This gives two cases to prove, the basis and step:

>> basis:
   "!n p. 0 + (n + p) = (0 + n) + p"

>> induction step:
   "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p"
   Assuming
    The induction hypothesis: "!n p. m + (n + p) = (m + n) + p"

The proof of the
>> basis:
   "!n p. 0 + (n + p) = (0 + n) + p"

is as follows:
>>>> This follows by using the equality,
     |- (0 + m = m) /\
        (m + 0 = m) /\
        ((SUC m) + n = SUC(m + n)) /\
        (m + (SUC n) = SUC(m + n))
     basic logical identities, and the assumptions made thus far.

This establishes

|- !n p. 0 + (n + p) = (0 + n) + p

The proof of the
>> induction step:
   "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p"
   Assuming
    The induction hypothesis: "!n p. m + (n + p) = (m + n) + p"

is as follows:
>>>> This follows by using the equality,
     |- (0 + m = m) /\
        (m + 0 = m) /\
        ((SUC m) + n = SUC(m + n)) /\
        (m + (SUC n) = SUC(m + n))
     basic logical identities, and the assumptions made thus far.

This establishes

!n p. m + (n + p) = (m + n) + p
|- !n p. (SUC m) + (n + p) = ((SUC m) + n) + p

This establishes
```

```
|- !m n p. m + (n + p) = (m + n) + p

This completes the proof of the conjecture
>> ADD_ASSOC:
   "!m n p. m + (n + p) = (m + n) + p"
```

In other words, all the information that is implicit or ephemeral in the interactive proof session, or simply bound to ML identifiers in an *ad hoc* way, is now explicitly structured and saved. Because it is saved, it can be printed in a readable form for later inspection and study.

To produce the account shown, the whole tree structure of intermediate subgoals, proof steps and and achieving theorems that is generated when a tactic is applied to a goal is preserved in an internal form. Thus, the meaning of names such as `ADD_CLAUSES` can be shown; separate branches of the tree (such as the branching into two cases that is caused by the induction proof step) are shown individually, even when the tactic is not phrased that way. After the printing of one branch of the tree, a reminder can be given of the next pending branch. ML identifiers such as '`INDUCT_TAC`' are referred to by their meaning and effect rather than simply by name. Most importantly, the account avoids using HOL-specific terminology or concepts. For example, reference is avoided to goals and subgoals, current assumptions, tactics, and rewrite rules.

The account is produced in the following way: First, the ML type of a goal is modified to include more information, such as a name for each assumption of the goal, and a name for the whole goal (useful when more than one subgoal is produced at some stage). A new type, *account*, is introduced to represent subgoal-theorem trees. Justifications are reconceived as mapping lists of accounts (of subgoals) to an account (of the original goal). Next, the ML type of a tactic is modified to map a new type goal to a list of new type subgoals together with a new type justification. Finally, a suite of printing functions is written in ML to enable the subgoal-theorem trees to be output in an understandable format.

Further whole and partial examples of accounts occur throughout this paper.

3 The Extended ML Types

The accounts depicted in the previous chapter are based on more elaborate types of goals, tactics, and justifications than exist in HOL itself. The new types enable enough information to be stored during the performance of a goal oriented proof to generate a comprehensible explanation afterwards.

In the existing system, the following expressions introduce the types for justifications (proofs), goals and tactics, respectively:

```
lettype proof = thm list -> thm  ;;
lettype goal = term list # term;;
lettype tactic = goal -> ((goal list) # proof);;
```

A goal is a term together with a list of current assumptions; and a tactic maps a goal to a list of subgoals and a justification, where the justification maps the theorems achieving the subgoals to the theorem achieving the original goal.

For the purpose of producing accounts, a new type, *named goal*, is introduced (via a constructor function):

```
type named_goal =
 mk_named_goal of string # (string # bool # term) list # term;;
```

A named goal of the form `mk_named_goal`(*s,sbtl,t*) corresponds to an ordinary goal *tl,t*, where the list of third components of the elements of *sbtl* is simply *tl*. That is, each assumption of a named goal is accompanied by a name (i.e. a string) and a boolean value (whose purpose is explained later); and each goal itself has a name (a string). The names are used in the printing of accounts to identify certain assumptions, and to distinguishe among multiple subgoals.

To specify the structure of an account, we first introduce a type for *proof steps*:

```
lettype proof_step =
 string # term list # thm list;;
```

The string part of a proof step identifies the function comprising the step (that is, a tactic or a tactic-valued function); while the lists of terms and theorems allow for parameters to be recorded (in case the function comprising

the step is not a tactic but a function mapping a term to a tactic, a theorem to a tactic, etc).

An account is defined recursively as consisting of a proof step (which acts on a goal), together with a list of the named subgoals induced by that step; a list of sub-accounts of the respective subgoals; and a theorem (purporting to achieve the original goal):

```
rectype named_account =
 mk_node of proof_step # (named_account list) # (named_goal list) # thm;;
```

An auxiliary function `extract_theorem` selects the theorem component of an account. It is defined by:

```
let extract_theorem ac =
 let mk_node(ps,al,gl,th) = ac in th;;
```

The relation of achievement between a theorem and a goal is the same here as in HOL.

In the new scheme, a justification (*named proof*) function simply maps a list of (sub)accounts back to an account:

```
lettype named_proof =
 (named_account)list -> named_account;;
```

This subsumes the justification in the HOL sense since each account includes a theorem (as its fourth component).

A tactic, finally, maps a named goal to a list of named subgoals and a justification function:

```
lettype named_tactic =
 named_goal -> (named_goal list) # named_proof;;
```

The recursiveness of accounts means that an account is a tree structure. This gives an explicit internal representation of the subgoal-proof tree associated with a goal oriented proof. A readable version then can be produced by a suite of print functions. These can be arbitrarily sophisticated – for example, choosing to present only 'important' proof steps, and doing so using natural language expertise. However, we consider only a simple presentation in this paper, presenting every proof step, and doing so using unvarying, stored phrases. Even so the print functions are rather complicated.

There are two modes of printing named goals, one for goals which are either one of several goals to be printed together, or are the initial goals in a proof; and one for solitary and non-initial goals. In either case, goals are identified with the symbol >>. The term is printed first (using HOL's function for printing terms); then the labelled assumptions are announced and printed (using HOL's string and term printing functions).

For example, the following is a named goal whose account was displayed in the previous chapter ():

```
mk_named_goal(‘ADD_ASSOC‘, [], "!m n p. m + (n + p) = (m + n) + p")
```

The induction tactic was applied to this goal to yield two subgoals. The induction step subgoal is printed as follows, since it is one of two subgoals produced at once:

```
>> induction step:
   "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p"
   Assuming
    The induction hypothesis: "!n p. m + (n + p) = (m + n) + p"
```

Printed as an only, non-initial goal it would look the same but without the name of the goal:

```
>> "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p"
   Assuming
    The induction hypothesis: "!n p. m + (n + p) = (m + n) + p"
```

There are also two modes of printing proof steps: one for steps which advance a goal and one for steps which solve it. In either case, proof steps are identified by the symbol >>>>. The function which prints a proof step looks up the string identifying the step. This produces the appropriate phrases for explaining that step. The elements lists of term and theorem parameters may appear in the printed result. For example, the induction step of the proof in question is

```
(‘NAMED_INDUCT_TAC‘, ["m"], [])
```

and that step is presented as follows, including the term parameter m:

```
>>>> The proof is by mathematical induction on "m".
     This gives two cases to prove, the basis and step:
```

This step advances rather than solves the goal, and is worded accordingly. In contrast, the induction case is subsequently solved by applying a rewriting tactic which uses any relevant current assumptions as well as an existing theorem of arithmetic. The rewriting proof step is

```
(`NAMED_ASM_REWRITE_TAC`,
 [],
 [|- (0 + m = m) /\
     (m + 0 = m) /\
     ((SUC m) + n = SUC(m + n)) /\
     (m + (SUC n) = SUC(m + n))])
```

and is printed as follows, including the theorem parameter shown:

```
>>>> This follows by using the equality,
     |- (0 + m = m) /\
        (m + 0 = m) /\
        ((SUC m) + n = SUC(m + n)) /\
        (m + (SUC n) = SUC(m + n))
     basic logical identities, and the assumptions made thus far.
```

An account is presented (recursively) relative to a goal. Given a goal and an account of its proof, the print function first prints the proof step component of the account (i.e. the top node of the subgoal-proof tree). That is either the only node of the tree (meaning that the goal was solved in one step) or not (meaning that the goal is just advanced by the step); the appropriate mode is thus selected for printing the proof step.

Second, the subgoal list component of the account is printed. Depending on whether the list contains just one or more than one subgoal, the appropriate mode is selected for printing the element(s) of the subgoal list.

Third, the subaccounts are printed (recursively), relative to the respective subgoals. This is accomplished by announcing, for each subgoal-subaccount pair, that the proof of the subgoal is about to follow; then printing the subgoal followed by the subaccount. (Where there is only one such pair, the announcement and the repeated printing of the subgoal are omitted.)

Finally, the theorem achieving the original goal is announced and printed. Where the theorem does not in fact achieve the goal, a message to that effect is also printed; an example of this contingency is shown in ().

In the example case, the original goal is

```
mk_named_goal(`ADD_ASSOC`, [], "!m n p. m + (n + p) = (m + n) + p")
```

and the internal representation of the whole account is

```
mk_node((`NAMED_INDUCT_TAC`, ["m"], []),
        [mk_node((`NAMED_ASM_REWRITE_TAC`,
                  [],
                  [|- (0 + m = m) /\
                      (m + 0 = m) /\
                      ((SUC m) + n = SUC(m + n)) /\
                      (m + (SUC n) = SUC(m + n))]),
                 [],
                 [],
                 |- !n p. 0 + (n + p) = (0 + n) + p);
         mk_node((`NAMED_ASM_REWRITE_TAC`,
                  [],
                  [|- (0 + m = m) /\
                      (m + 0 = m) /\
                      ((SUC m) + n = SUC(m + n)) /\
                      (m + (SUC n) = SUC(m + n))]),
                 [],
                 [],
                 . |- !n p. (SUC m) + (n + p) = ((SUC m) + n) + p)],
        [mk_named_goal(`basis`, [], "!n p. 0 + (n + p) = (0 + n) + p");
         mk_named_goal(`induction step`,
                       [(`induction hypothesis`,
                         true,
                         "!n p. m + (n + p) = (m + n) + p")],
                       "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p")],
        |- !m n p. m + (n + p) = (m + n) + p)
```

The whole account is thus printed as follows:

```
>>>> The proof is by mathematical induction on "m".
     This gives two cases to prove, the basis and step:

>> basis:
   "!n p. 0 + (n + p) = (0 + n) + p"

>> induction step:
   "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p"
   Assuming
    The induction hypothesis: "!n p. m + (n + p) = (m + n) + p"

The proof of the
>> basis:
   "!n p. 0 + (n + p) = (0 + n) + p"

is as follows:
>>>> This follows by using the equality,
     |- (0 + m = m) /\
        (m + 0 = m) /\
        ((SUC m) + n = SUC(m + n)) /\
        (m + (SUC n) = SUC(m + n))
     basic logical identities, and the assumptions made thus far.
```

```
This establishes

|- !n p. 0 + (n + p) = (0 + n) + p

The proof of the
>> induction step:
   "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p"
   Assuming
    The induction hypothesis: "!n p. m + (n + p) = (m + n) + p"

is as follows:
>>>> This follows by using the equality,
     |- (0 + m = m) /\
        (m + 0 = m) /\
        ((SUC m) + n = SUC(m + n)) /\
        (m + (SUC n) = SUC(m + n))
     basic logical identities, and the assumptions made thus far.

This establishes

!n p. m + (n + p) = (m + n) + p
|- !n p. (SUC m) + (n + p) = ((SUC m) + n) + p

This establishes

|- !m n p. m + (n + p) = (m + n) + p
```

In contexts in which an account to be printed is a top level account rather than a subaccount of another, a prologue and epilogue are printed around the rest of the printout. Here, the prologue is

```
This is the proof of the conjecture
>> ADD_ASSOC:
   "!m n p. m + (n + p) = (m + n) + p"
```

and the epilogue is

```
This completes the proof of the conjecture
>> ADD_ASSOC:
   "!m n p. m + (n + p) = (m + n) + p"
```

This produces the whole account shown in the previous chapter. The algorithm described for printing accounts determines the order in which the nodes of the subgoal-proof tree are printed: the tree is traversed depth first and left to right. This method of printing a (tree-structured) account has the advantage of producing a 'flat' result rather than a result mirroring the tree structure by use of indentation or other device, which is useful, as the accounts can be indefinitely deep. The method also maintains indicators of the original tree structure by repeating each subgoal before giving its account, where there is more than one subgoal to be presented.

Further examples of printed accounts are shown throughout the paper.

4 Elementary Tactics

For the purpose of generating proof accounts, the tactics provided in HOL can be represented by three groups of corresponding named tactics:

1. Simple tactics which mirror the corresponding standard tactics, merely elaborating them with names for their relevant values;

2. Complex tactics which use the corresponding standard tactics, but which then further process the results into more meaningful formats;

3. Tactics whose relation to natural patterns of reasoning is distant, and for which generating accounts raises philosophical problems; these cannot be implemented along the lines of the corresponding standard tactics.

This section and Section (...) address the first group; Sections (...), (...) and (...) address the second group; and Sections (...), (...) and (...) address the third.

In producing an account of the application of a tactic to a goal, it is useful to know something about the possible outcomes of the application. A particular tactic, when applied to a goal, either produces some number of subgoals (together with a justification), or else it raises an exception (i.e. fails). Where a tactic succeeds on a goal, the number of subgoals produced may be fixed for the tactic, or it may vary indefinitely, depending on the goal. Some tactics have the capacity to *solve* goals; i.e. to produce no subgoals (together with an appropriate justification). Other tactics are able to *advance* goals (i.e. to produce one or more subgoals); some can do either. Finally, a tactic that advances a goal can do so by producing subgoals either with changed lists of assumptions, or with changed terms – or both.

Based on the possible outcomes of applying a tactic to a goal, a scheme for comprehensibly presenting the proof step it represents, and the subgoals it induces, can be designed. The treatment of a few simple tactics illustrates the methods and the range of issues involved.

4.1 The Implementation of Named Tactics: (GEN_TAC)

In this section, we sketch the way in which simple named tactics are implemented to produce accounts. `GEN_TAC` is used as an example.

The tactic GEN_TAC maps a goal with a universally quantified term (i.e. a term of the form !x.t[x] to a list with just one subgoal, whose term is instantiated to the bound variable (or, if necessary, a fresh variable not free anywhere in the goal). That is, the new term is of the form t[x']. GEN_TAC fails on goals whose terms are not universally quantified; where it succeeds it produces a subgoal list of fixed length (one). GEN_TAC changes the term of a goal, where it succeeds, but never the assumption list. It cannot solve a goal, but only advance one.

The use of GEN_TAC is illustrated in the example below. Two new predicates, DIVIDES and PRIME, are defined here:

```
DIVIDES_DEF = |- !m n. m DIVIDES n = ~(m = 0) /\ (?q. q * m = n)

PRIME_DEF =
|- !n. PRIME n = n > 1 /\ (!m. m DIVIDES n ==> (m = 1) \/ (m = n))
```

Suppose that a goal, g, is introduced, as shown below, and that GEN_TAC is applied to g to give a list (gl1) of one subgoal, and a justification function (p1):

```
let g = [], "!n. (n > 1) ==> (?p. (PRIME p) /\ (p DIVIDES n))";;
g = ([], "!n. n > 1 ==> (?p. PRIME p /\ p DIVIDES n)")

#let gl1,p1 = GEN_TAC g;;
gl1 = [([], "n > 1 ==> (?p. PRIME p /\ p DIVIDES n)")] : goal list
p1 = - : proof
```

Given, eventually, the theorem th

```
th = |- n > 1 ==> (?p. PRIME p /\ p DIVIDES n)
```

the function p maps th to a theorem achieving g:

```
#p1[th];;
|- !n. n > 1 ==> (?p. PRIME p /\ p DIVIDES n)
```

To produce an account of this goal-oriented proof, a corresponding new tactic, called NAMED_GEN_TAC is defined. NAMED_GEN_TAC maps the corresponding named goal to a list of one named goal, together with a named proof (the justification). The justification, in turn, maps a list of one *account* (the account of the one subgoal) to another account (the account of the original goal). To define NAMED_GEN_TAC, given an arbitrary named goal, requires (i) the subgoal to be constructed and (ii) the justification to be specified. The corresponding named goal (ng), called 'example_1', is:

```
#let ng =
mk_named_goal(‘example_1‘,
              [],
              "!n. n > 1 ==> (?p. PRIME p /\ p DIVIDES n)")
```

Since it is easy to extract an ordinary goal from a named goal, the effect of the ordinary tactic `GEN_TAC` on the corresponding ordinary goal can be computed; this gives an ordinary subgoal and justification (as shown earlier). To then construct the named subgoal using the ordinary subgoal is very simple, since the (named and flagged) assumptions of the original named goal should not be changed by application of `NAMED_GEN_TAC`. The name of the subgoal does not matter, since it is an only subgoal, so the name of the original goal is used, arbitrarily, as the subgoals's name. The term of the subgoal is just the term of the ordinary subgoal. Thus the list of subgoals produced by `NAMED_GEN_TAC` on the named goal (`ng`) is:

```
[mk_named_goal(‘example_1‘,
               [],
               "n > 1 ==> (?p. PRIME p /\ p DIVIDES n)")]
```

Computing the justification for a named goal is also straightforward. A function is defined which maps a list containing one account (the account of the named subgoal) to a new account (the account of the named goal). That is, the new justification is specified as a function of the form

```
\[ac:named account]. mk_node(..., ..., ..., ...)
```

with the parameter `ac` representing the account of the subgoal, and the four slots representing the following components:

1. The proof step;
2. The list containing the sub-account of the subgoal
3. The list containing the subgoal, and
4. The theorem that achieves the subgoal.

The proof step consists of a string, to identify the tactic applied, a list of any term parameters to be remembered, and a list of any theorem parameters. `NAMED_GEN_TAC` (like `GEN_TAC`) does not involve theorem parameters, but does

involve a term: the term which is instantiated. To identify the proof step, the string ‘NAMED_GEN_TAC‘ will do. The new subgoal is known (as explained above), so the third component is easy. The list containing the account (ac) of the subgoal is supplied to the justification (via the lambda binding), so this gives the fourth item. Finally, the justification of the ordinary GEN_TAC has already been computed. From the account of the new subgoal, the theorem achieving the new subgoal can be extracted (it is the fourth component of the account); then the ordinary justification can be applied to that theorem to produce the theorem achieving the main goal. Thus the new justification is denoted by the expression

```
\[ac]. mk_node((‘NAMED_GEN_TAC‘, ["n:num"], []),
               [ac],
               [mk_named_goal(‘example_1‘,
                              [],
                              "n > 1 ==> (?p. PRIME p /\ p DIVIDES n)")],
                p1[extract_theorem ac])
```

When the named tactic is applied to the named goal, a list of named subgoals and a named proof (justification) result:

```
#NAMED_GEN_TAC ng;;
([mk_named_goal(‘example_1‘,
                [],
                "n > 1 ==> (?p. PRIME p /\ p DIVIDES n)")],
 -)
: (named_goal list # named_proof)
```

When the account of the original goal ng is finally produced – by applying the named proof to the actual account of the subgoal – it is of the form

```
mk_node((‘NAMED_GEN_TAC‘, ["n"], []),
        [mk_node($\cdots$)],
        [mk_named_goal(‘example_1‘,
                       [],
                       "n > 1 ==> (?p. PRIME p /\ p DIVIDES n)")],
        |- !n. n > 1 ==> (?p. PRIME p /\ p DIVIDES n))
: named_account
```

where the “. . .” representing the actual account of the subgoal may be arbitrarily complex.

This internal representation is made readable by a suite of printing functions which (i) produce a linear layout, and (ii) use the strings recording

proof steps to look up a packaged 'explanation' of the strategy behind the tactic.

To print the account of a named goal, the proof step is first announced and printed; then the subgoals are announced and printed; then the account of each subgoal is announced and (recursively) printed; and finally, the theorem achieving the original goal is announced and printed. To print a proof step requires a print function defined as a large conditional with a branch for each possible string which identifies a proof step. The print function provides a natural wording for the step denoted by the string – that is, it describes the natural pattern of reasoning implemented by the tactic behind the step. To print a goal involves identifying and printing the term of the goal, and then identifying and printing the assumptions.

The printed form of the example account is shown (partially) below. (As we have not said anything about the proof of the subgoal, the the "..." represents the printout of the account of the subgoal.)

```
>>>> Consider an arbitrary "n":
     We show:
>> "n > 1 ==> (?p. PRIME p /\ p DIVIDES n)"
...
This establishes
|- n > 1 ==> (?p. PRIME p /\ p DIVIDES n)
This establishes
|- !n. n > 1 ==> (?p. PRIME p /\ p DIVIDES n)
```

The string 'GEN_TAC' is used to generate the wording at "Consider an arbitrary ..." (and the term remembered then appears). The wording suggests the natural pattern of reasoning in something like the way that a textbook might put it. If an account to be printed is the outermost account of a particular proof, a prologue and epilogue are added around its printout:

```
This is the proof of the conjecture
>> example_1:
   "!n. n > 1 ==> (?p. PRIME p /\ p DIVIDES n)"
...
This completes the proof of the conjecture
>> example_1:
   "!n. n > 1 ==> (?p. PRIME p /\ p DIVIDES n)"
```

The internal form of the account could be rendered in many other ways, of course, each with its own suite of print functions. The particular suite that has been implemented reports every proof step in detail, and uses the format shown.

Most of the named tactics corresponding to simple HOL tactics are implemented similarly way to `GEN_TAC`.

4.2 Solving a Goal: ACCEPT_TAC

Several tactics are capable, unlike `GEN_TAC`, of solving goals. The simplest of these is `ACCEPT_TAC`, which in fact *only* solves, and cannot advance, goals. `ACCEPT_TAC` is a function which maps a theorem to a tactic which when applied to a goal either produces an empty list of subgoals, or else fails. The former happens iff the conclusion of the theorem is the same as the term of the goal (up to alpha-conversion); in that case, the justification of `ACCEPT_TAC`, applied to the corresponding empty list of theorems, produces the same theorem as provided to `ACCEPT_TAC`. This is demonstrated by solving the following goal (which might, perhaps be a case in a larger proof) using the pre-proved HOL theorem `MULT_SYM`:

```
#let g = ["x > 0";"y > 0"],"x * y = y * x";;
g = (["x > 0"; "y > 0"], "x * y = y * x") : goal

#MULT_SYM;;
|- !m n. m * n = n * m

#let thm = SPECL ["x:num";"y:num"] (MULT_SYM);;
thm = |- x * y = y * x

#let gl,p = ACCEPT_TAC thm g;;
gl = [] : goal list
p = - : proof

#p[];;
|- x * y = y * x
```

The account of this fragment of proof uses a wording to express the natural strategy behind `ACCEPT_TAC`. (In the corresponding named goal, the two assumptions are given names.)

```
This is the proof of the conjecture
>> example_2:
   "x * y = y * x"
   Assuming
```

```
    The fact1: "x > 0"
    The fact2: "y > 0"

>>>> The theorem
     |- x * y = y * x
     is proposed to satisfy this.

This establishes

|- x * y = y * x

This completes the proof of the conjecture
>> example_2:
   "x * y = y * x"
   Assuming
    The fact1: "x > 0"
    The fact2: "y > 0"
```

Note that `NAMED_ACCEPT_TAC` must record its theorem parameter in order that the account be understandable.

If the theorem to which the justification of `ACCEPT_TAC` is applied has an appropriate conclusion but fails to achieve the original goal through having hypotheses beyond the assumptions of the goal, then this failure is noted at the appropriate points in the account – here, in the prologue and epilogue. Suppose, for example, that we have proved the easy theorem `thm'`, as shown below, and that `thm'` is supplied to `NAMED_ACCEPT_TAC` in place of `thm`:

```
x = 3 |- x * y = y * x
```

In HOL, an empty list of subgoals would again ensue, but the justification would then produce the theorem `x = 3 |- x * y = y * x`. The account makes the nature of this failure clear:

```
This is the attempted proof of the conjecture
>> example_2:
   "x * y = y * x"
   Assuming
    The fact1: "x > 0"
    The fact2: "y > 0"

>>>> The theorem
     x = 3 |- x * y = y * x
     is proposed to satisfy this.

This establishes

x = 3 |- x * y = y * x

which does not satisfy
>> "x * y = y * x"
   Assuming
```

```
    The fact1: "x > 0"
    The fact2: "y > 0"
This completes the attempted proof of the conjecture
>> example_2:
   "x * y = y * x"
   Assuming
    The fact1: "x > 0"
    The fact2: "y > 0"
```

The wording seen in the prologue and epilogue are chosen by the print functions when the achievement failure is detected in the subgoal-proof tree being printed.

`NAMED_ACCEPT_TAC` is implemented similarly to `NAMED_GEN_TAC`, except that instead of constructing a list containing one subgoal, it simply returns an empty list of subgoals. The justification does not involve inference – as `GEN_TAC`'s does, but simply maps the empty list of theorems to the theorem provided. While the implementation of `NAMED_GEN_TAC` must remember a term, that of `NAMED_ACCEPT_TAC` must remember the theorem parameter to which it was applied.

4.3 Naming New Assumptions: DISCH_TAC

`DISCH_TAC`, like `GEN_TAC`, can advance but not solve goals; and where it succeeds, it produces exactly one subgoal. Unlike `GEN_TAC`, it not only changes the term of a goal, but also changes the assumption list (by adding a new assumption):

```
#let g = [],"n > 1 ==> (?p. PRIME p /\ p DIVIDES n)";;
g = ([], "n > 1 ==> (?p. PRIME p /\ p DIVIDES n)") : (* list # term)

#let gl,p = DISCH_TAC g;;
gl = [(["n > 1"], "?p. PRIME p /\ p DIVIDES n")] : goal list
p = - : proof
```

Once we have proved the theorem `th`

```
th = |- ?p. PRIME p /\ p DIVIDES n
```

we can then apply the justification (`p`) to yield the theorem achieving the original goal:

```
#p[th];;
|- n > 1 ==> (?p. PRIME p /\ p DIVIDES n)
```

The corresponding named tactic is implemented along the lines of the previous named tactics, except that it must in addition give a name to the added assumption to indicate that this assumption was, in a previous goal, the antecedent of an implication. The account constructed by the named tactic uses this name, and supplies a natural wording for the strategy, applied to the corresponding named goal:

```
This is the proof of the conjecture
>> example_3:
   "n > 1 ==> (?p. PRIME p /\ p DIVIDES n)"

>>>> It is sufficient to prove:

>> "?p. PRIME p /\ p DIVIDES n"
   Assuming
    The antecedent: "n > 1"

...

This establishes

n > 1 |- ?p. PRIME p /\ p DIVIDES n

This establishes

|- n > 1 ==> (?p. PRIME p /\ p DIVIDES n)

This completes the proof of the conjecture
>> example_3:
   "n > 1 ==> (?p. PRIME p /\ p DIVIDES n)"
```

4.4 Transforming Subgoals: SUBST1_TAC

The ML function `SUBST1_TAC`, like `ACCEPT_TAC`, maps a theorem to a tactic. The theorem must have a conclusion of equational form; it is is used to make and justify a substitution throughout the term of a goal for all free instances of the left hand side of the equation by the right hand side of the equation. Like `GEN_TAC`, a tactic of the form `SUBST1_TAC` *th*, where it succeeds, produces a subgoal list of fixed length one. Also like `GEN_TAC`, it advances but does not solve goals; and it transform the term of a goal but does not alter the assumptions. For example:

```
#let g = [],"!n:num. n > 1 ==> n DIVIDES n";;
g = ([], "!n. n > 1 ==> n DIVIDES n")
```

Suppose that the theorem `th` is an instance of the definition of division:

```
th = |- n DIVIDES n = ~(n = 0) /\ (?q. q * n = n)
```

Then substituting throughout the goal according to the specialized form of the definition, is a way of unfolding the goal into more basic terms:

```
#let gl1,p1 = (GEN_TAC THEN SUBST1_TAC th)g;;
gl1 = [([], "n > 1 ==> ~(n = 0) /\ (?q. q * n = n)")] : goal list
p1 = - : proof
```

The printed form of the account on the corresponding named goal explains the effect of `SUBST_TAC`:

```
This is the proof of the conjecture
>> example_4:
   "!n. n > 1 ==> n DIVIDES n"

>>>> Consider an arbitrary "n":
     We show:

>> "n > 1 ==> n DIVIDES n"

>>>> We substitute according to the following equality:
     |- n DIVIDES n = ~(n = 0) /\ (?q. q * n = n).
     Thus, it is sufficient to prove:

>> "n > 1 ==> ~(n = 0) /\ (?q. q * n = n)"

...

This establishes

|- n > 1 ==> ~(n = 0) /\ (?q. q * n = n)

This establishes

|- n > 1 ==> n DIVIDES n

This establishes

|- !n. n > 1 ==> n DIVIDES n

This completes the proof of the conjecture
>> example_4:
   "!n. n > 1 ==> n DIVIDES n"
```

It can be seen that the theorem parameter to `NAMED_SUBST_TAC` has to be remembered in order to explain fully the substitution – as for `NAMED_ACCEPT_TAC`. The implementation is similar to previous ones.

4.4.1 Implicit Assumptions from Invalid Proof Steps

Although `SUBST_TAC` is apparently straightforward, there is one difficulty that may arise. To explain it, we use the arithmetic constants `SUC` and `PRE`, provided in HOL, for the successor and predecessor functions (respectively) on

the natural numbers. The predecessor function is characterized by the theorem

```
|- (PRE 0 = 0) /\ (!m. PRE(SUC m) = m)
```

and about the successor we know that

```
|- !n. ~(SUC n = 0)
```

Suppose that the goal is to prove the following (for any `x`)

```
#let g = [], "PRE(SUC(PRE x)) = PRE x";;
g = ([], "PRE(SUC(PRE x)) = PRE x") : (* list # term)
```

and that we have already proved the theorem `th`:

```
~(x = 0) |- SUC(PRE x) = x
```

(which is not difficult to prove).

If a user unwittingly were to try to procede in HOL by making a substitution based on the theorem `th`, the resulting subgoal would appear *without* recording the fact that an assumption (`~(x = 0)`) had thereby been introduced. The result would appear to be as hoped:

```
#let gl1,p1 = SUBST1_TAC th g;;
gl1 = [([], "PRE x = PRE x")] : goal list
p1 = - : proof
```

This subgoal could then be solved by appeal to reflexivity:

```
#let th' = REFL "PRE x";;
 th' = |- PRE x = PRE x

#let gl2,p2 = ACCEPT_TAC th' (hd gl1);;
gl2 = [] : goal list
p2 = - : proof

#let th2 = p2[];;
th2 = |- PRE x = PRE x
```

This still appears to solve the problem; it leads to a theorem which achieves the one subgoal in `gl1`. However, the justification of the substitution (`p1`) maps `th2` to a theorem `th1`

```
#let th1 = p1[th2];;
th1 = . |- PRE(SUC(PRE x)) = PRE x

#print_all_thm th1;;
~(x = 0) |- PRE(SUC(PRE x)) = PRE x
```

which, because it is contingent on some hypothesis, does *not* achieve the original goal. This sudden failure of achievement is the first indication to the user that an assumption has been introduced 'behind the scenes' – as a result of the theorem parameter to `SUBST1_TAC` having depended on the hypothesis `~(x = 0)`. The cause of the failure may not be immediately apparent – even after the hypothesis (printed by default as a dot) is examined.

Indeed, if instead of `th'` (`|- PRE x = PRE x`) we had proved an easy theorem `th''`

```
~(x = 0) |- PRE x = PRE x
```

and we had supplied `th''` rather than `th'` as the solution of the subgoal in `gl1` (i.e. `([], "PRE x = PRE x")`), then the theorem (`th2`) which was supplied to the justification (`p1`) of the substitution would already depend on the hypothesis `~(x = 0)`:

```
#let gl2,p2 = ACCEPT_TAC th'' (hd gl1);;
gl2 = [] : goal list
p2 = - : proof

#let th2 = p2[];;
th2 = . |- PRE x = PRE x

#print_all_thm th2;;
~(x = 0) |- PRE x = PRE x

#let th1 = p1[th2];;
th1 = . |- PRE(SUC(PRE x)) = PRE x

#print_all_thm th1;;
~(x = 0) |- PRE(SUC(PRE x)) = PRE x
```

As can be seen, the end result is the same as before. That is, the justification of the substitution (the function `p1`) "knows" about the invisible assumption `~(x = 0)`, so whether the justification is applied to the theorem with conclusion `PRE x = PRE x`) with the hypothesis `~(x = 0)` or without the hypothesis makes no difference; in either case, the result is a theorem *with* the hypothesis `~(x = 0)`. However, what the justification function "knows" is not readily apparent to a user. We will take this behaviour of the justification to be the

criterion of whether a tactic applied to a given goal introduces an *implicit assumption*.

To see why the justification necessarily adds the hypothesis to the theorem it returns, where it is lacking, one must examine the inference rule for substitution which supports the substitution tactic. The rule specifies that in using an established equality to substitute equals for equals throughout the conclusion of a given theorem, the hypotheses of the equality theorem, as well as the hypotheses of the theorem into which the substitution is made, are propagated through to the resulting theorem. (See ref, Ch 3.)

```
A |- t = u    B |- P(t)
----------------------
     A u B |- P(u/t)
```

In general, a tactic is called *invalid* if it is able to generate, on some goal, subgoals and a justification such that achieving the subgoals does not necessarily entail, via the justification, achieving the goal. Invalidity necessarily characterizes any tactic constructed by applying a function of type `thm -> tactic` (or `thm list -> tactic`), etc, to appropriate values to create a tactic. The property thereby pertains to quite a few of the commonly used HOL tactics, including `DISCH_TAC`, which was described earlier, as well as `SUBST1_TAC`. (See appendix ... listing all such HOL tactics.)

Any of these invalid tactics can be applied validly or invalidly to goals. `SUBST1_TAC th`, for example, was applied invalidly to the goal `g`, in the last example, because `g` included no assumptions – in particular, it did not include as an assumption the hypothesis `~(x = 0)` of the substitution theorem `th`.

4.4.2 Implicit Assumptions without Use

In the previous example, the theorem `th` was *used* in the substitution step; it may appear that that is essential for the hypothesis `~(x = 0)` to have been made implicitly. However, this is not so. Another example of the same sort illustrates the subtle point that the appearance of the invisible assumption does not depend on the theorem with the hypothesis having had an effect on the goal. In the following example, the attempted substitution has no effect, because there is no suitable substitution instance for the term `SUC(PRE x)`. Suppose the goal is

```
#let g = [],"PRE(SUC x) = x";;
g = ([], "PRE(SUC x) = x") : (* list # term)
```

and the same substitution theorem, `th` (`~(x = 0) |- SUC(PRE x) = x`), is engaged (but to no effect):

```
#let gl,p = SUBST1_TAC th g;;
gl = [([], "PRE(SUC x) = x")] : goal list
p = - : proof
```

Once the single subgoal is achieved – without our specifying how – by a theorem `th'`

```
#th';;
|- PRE(SUC x) = x
```

the justification can be applied to give a result:

```
#p[th'];;
. |- PRE(SUC x) = x

#print_all_thm it;;
~(x = 0) |- PRE(SUC x) = x
```

Thus, despite the fact that the theorem `th'` itself achieves the subgoal

```
[], "PRE(SUC x) = x")
```

and the fact that the substitution tactic has had no effect, the justification (`p`) of the substitution tactic *still* produces a theorem depending on the hypothesis `~(x = 0)`. That is, the substitution step necessarily introduces an assumption behind the scenes – by embedding that hypothesis in the function that justifies the (effective or ineffective) substitution step.[11]

4.4.3 Implicit Assumptions from Valid Proof Steps

Although the appearance of the unexpected hypothesis in the previous two sections was caused by an invalid use of a tactic, the introduction of invisible assumptions does not arise *only* through invalidity – the mechanism is actually more subtle still. We return to the first substitution example (Section 4.4.1) to illustrates the same effect, but without the invalid use of tactics and without failing to achieve the original goal.

Suppose we refer to the same theorem `th`:

[11] Possibly, HOL's substitution tactic could be implemented so that if it detected that it has had no effect it would return a justification that did not rely on the inference rule for substitution – which is the origin of the hypothesis of the result. However, this would be complicated, probably inefficient, and would have to be done for quite a few other similarly constructed tactics.

```
~(x = 0) |- SUC(PRE x) = x
```

and this time use a goal resembling that of Section 4.4.1, but which includes the assumption in question to begin with:

```
#let g = ["~(x = 0)"], "(PRE(SUC(PRE x)) = PRE x)";;
g = (["~(x = 0)"], "PRE(SUC(PRE x)) = PRE x") : goal
```

The use of the substitution tactic is now valid:

```
#let gl1,p1 = SUBST1_TAC th (hd gl1);;
gl1 = [(["~(x = 0)"], "PRE x = PRE x")] : goal list
p1 = - : proof
```

If the theorem `th'` (as in Section 4.4.1)

```
th' = |- PRE x = PRE x
```

or indeed `th''` (also as in Section 4.4.1)

```
~(x = 0) |- PRE x = PRE x
```

is now supplied as the solution to the goal in `gl1`, the justification (`p1`) of the substitution – as before – produces a theorem (`th2`) that depends on the condition `~(x = 0)`. (This time, though, the resulting theorem *does* achieve the goal `g`.)

```
#let gl2,p2 = ACCEPT_TAC th' (hd gl2);;
gl2 = [] : goal list
p2 = - : proof

#let th2 = p2[];;
th2 = |- PRE x = PRE x

#let th1 = p1[th2];;
th1 = . |- PRE(SUC(PRE x)) = PRE x

#print_all_thm th1;;
~(x = 0) |- PRE(SUC(PRE x)) = PRE x
```

The dependence on `~(x = 0)` happens despite the validity of the substitution on the subgoal – that is, despite the fact that at the point where the substitution tactic was applied, the condition `~(x = 0)` was already a standing assumption. (This is *not* an automatic effect of `~(x = 0)` having already been an assumption – not all assumptions reappear thus.) The implicit assumption introduced by the tactic manifests itself in the effect of the justification function of that tactic, and for exactly the same reason as in the previous two examples: the propagation of assumptions in the inference rule for substitution.

4.4.4 Accounting for Implicit Assumptions

The first and second examples (in Sections 4.4.1 and 4.4.2 respectively), involving invalid reasoning, might be dismissed simply as poor HOL style; indeed, such reasoning is precluded by the HOL subgoal interface in its most restrictive mode. However, in the third example, the reasoning is completely valid, and the example in fact illustrates a commonly used method in HOL tactical proof. There are, in addition, several other (valid) ways in which assumptions can be caused to appear behind the scenes, and these likewise cannot be dismissed as poor HOL style – they are features of HOL's current design. (These other ways are discussed in ... .) For all of these cases, it is necessary, in proof accounts, to deal with the issue of implicit assumptions.

The accounting method we propose is to record *all* assumptions that pertain to a goal, whether or not they would be visible ordinarily. Implicit assumptions are identified by the boolean value `false`; this is the purpose of the boolean component of an assumption of a named goal. Whether or when implicit assumptions are printed is a feature of a particular printing routine, but the information is anyway available to print. (Currently, they are always printed.)

With implicit assumptions recorded in accounts, the invalid use of substitution seen above in the first (invalid) example (Section 4.4.1) – which might well have puzzled the user – is accounted for as follows:

```
This is the attempted proof of the conjecture
>> example_5:
   "PRE(SUC(PRE x)) = PRE x"

>>>> We substitute according to the following equality:
     ~(x = 0) |- SUC(PRE x) = x.
     Thus, it is sufficient to prove:

>> "PRE x = PRE x"
   Assuming implicitly
    The hypothesis of the equality: "~(x = 0)"

>>>> The theorem
     |- PRE x = PRE x
     is proposed to satisfy this.

This establishes

|- PRE x = PRE x

This establishes

~(x = 0) |- PRE(SUC(PRE x)) = PRE x

which does not satisfy
```

```
>> "PRE(SUC(PRE x)) = PRE x"

This completes the attempted proof of the conjecture
>> example_5:
   "PRE(SUC(PRE x)) = PRE x"
```

This account clears up all the mystery from the situation: first, the subgoal decomposition records the introduced assumption so that it can be seen from the point at which it becomes an assumption onward; second, the transition (via the justification of the substitution tactic) from the establishment of the theorem `|- PRE x = PRE x` to the theorem `~(x = 0) |- PRE(SUC(PRE x)) = PRE x` can be understood by reference to the implicit assumption of the relevant subgoal; and finally, the failure to achieve the original goal (because of the additional hypothesis) is noted and made clear.

The account of the second example (Section 4.4.2), in which the (invalid) substitution step has no effect on the term of the goal, makes clear that the step does have the side effect of introducing an implicit assumption, which later manifests itself in the chain of achieving theorems produced by successive justifications:

```
This is the attempted proof of the conjecture
>> example_6:
   "PRE(SUC x) = x"

>>>> We substitute according to the following equality:
     ~(x = 0) |- SUC(PRE x) = x.
     Thus, it is sufficient to prove:

>> "PRE(SUC x) = x"
   Assuming implicitly
    The hypothesis of the equality: "~(x = 0)"

...

This establishes

|- PRE(SUC x) = x

This establishes

~(x = 0) |- PRE(SUC x) = x

which does not satisfy
>> "PRE(SUC x) = x"

This completes the attempted proof of the conjecture
>> example_6:
   "PRE(SUC x) = x"
```

The account produced for the third (valid) example (Section 4.4.3), in which the assumption `~(x = 0)` belongs to the goal at the point where the

substitution is made, is again intended to clear up any mystery about the reappearance of the implicit assumption in the chain of achieving theorems:

```
This is the proof of the conjecture
>> example_7:
   "PRE(SUC(PRE x)) = PRE x"
   Assuming
    The fact: "~(x = 0)"

>>>> We substitute according to the following equality:
     ~(x = 0) |- SUC(PRE x) = x.
     Thus, it is sufficient to prove:

>> "PRE x = PRE x"
   Assuming
    The fact: "~(x = 0)"
   Assuming implicitly
    The hypothesis of the equality: "~(x = 0)"

>>>> The theorem
     |- PRE x = PRE x
     is proposed to satisfy this.

This establishes

|- PRE x = PRE x

This establishes

~(x = 0) |- PRE(SUC(PRE x)) = PRE x

This completes the proof of the conjecture
>> example_7:
   "PRE(SUC(PRE x)) = PRE x"
   Assuming
    The fact: "~(x = 0)"
```

The explicit assumption `~(x = 0)`, in the subgoal

```
>> "PRE x = PRE x"
   Assuming
    The fact: "~(x = 0)"
   Assuming implicitly
    The hypothesis of the equality: "~(x = 0)"
```

does not explain the dependence on `~(x = 0)` of the corresponding achieving theorem

```
~(x = 0) |- PRE(SUC(PRE x)) = PRE x
```

The noting of the introduction of the implicit assumption, in each of the accounts of substitution, is achieved by implementing `NAMED_SUBST_TAC` so that whenever it is applied to a theorem, and the resulting tactic to a goal,

any hypotheses of the theorem are recorded as implicit assumptions of the subgoal being constructed. Any such assumption is labelled to indicate its origin – in the case of substitution with the string

```
`the hypothesis of the equality`
```

and with the boolean value `false` to indicate that it is an *implicit* assumption. The only futher care required is that in extracting an ordinary goal from a named goal (so that the results of the ordinary `SUBST1_TAC` can be computed), only explicit assumptions should be included; assumptions of the named goal labelled with `false` are ignored. Implicit assumptions are included again, however, in the named subgoal being constructed by `NAMED_SUBST_TAC` – that is, implicit assumptions persist from named goals to named subgoals, as one would expect.

The printing routine for goals is then arranged to print explicit and implicit assumptions separately (as illustrated in the accounts above). The routine for printing whole accounts is arranged to produce an appropriate message (again, as illustrated) when a candidate theorem fails to achieve the subgoal for which it was intended; and when the theorem purporting to do so fails to achieve an initial (outermost) goal.

4.5 Multiple Subgoals: INDUCT_TAC

The numerical induction tactic is an example of a tactic which produces more than one subgoal – it always produces one basis and and one step case, when it succeeds at all. In both subgoals, there is a transformed term; and in the step goal, there is a different assumptions list – a new assumption (the induction hypothesis) is added. For example, the proof of the associativity of addition (normally pre-proved in HOL) is by induction:

```
#let g = [],"!m n p. m + (n + p) = (m + n) + p";;
g = ([], "!m n p. m + (n + p) = (m + n) + p")

#let [g1;g2],p = INDUCT_TAC g;;
g1 = ([], "!n p. 0 + (n + p) = (0 + n) + p") : goal
g2 =
(["!n p. m + (n + p) = (m + n) + p"],
 "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p")
: goal
p = - : proof
```

The corresponding named goal is

```
mk_named_goal(‘example_8‘, [], "!m n p. m + (n + p) = (m + n) + p")
```

To specify the corresponding named tactic `NAMED_INDUCT_TAC` requires constructing the two *named* subgoals from the two ordinary subgoals. This in turn requires naming each subgoal, and naming the new assumption of the step subgoal. The named justification is constructed much as for the previous tactics. Here, it is a function that maps a list of two sub-accounts to an account of the original goal. The string ‘`NAMED_INDUCT_TAC`‘ identifies the tactic used, and the induction variable (`m`) is recorded. When the whole proof is completed and printed, the induction is accounted for as follows:

```
This is the proof of the conjecture
>> example_8:
   "!m n p. m + (n + p) = (m + n) + p"

>>>> The proof is by mathematical induction on "m".
     This gives two cases to prove, the basis and step:

>> basis:
   "!n p. 0 + (n + p) = (0 + n) + p"

>> induction step:
   "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p"
   Assuming
    The induction hypothesis: "!n p. m + (n + p) = (m + n) + p"

The proof of the
>> basis:
   "!n p. 0 + (n + p) = (0 + n) + p"

is as follows:

...

This establishes

|- !n p. 0 + (n + p) = (0 + n) + p

The proof of the
>> induction step:
   "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p"
   Assuming
    The induction hypothesis: "!n p. m + (n + p) = (m + n) + p"

is as follows:

...

This establishes

!n p. m + (n + p) = (m + n) + p
|- !n p. (SUC m) + (n + p) = ((SUC m) + n) + p
```

```
This establishes

|- !m n p. m + (n + p) = (m + n) + p

This completes the proof of the conjecture
>> example_8:
   "!m n p. m + (n + p) = (m + n) + p"
```

In printing this account, the accounts of the two subgoals are printed in the order in which the subgoals were announced. Since there is more than one subgoal, and the account of each can be arbitrarily long, each sub-account is prefaced by a reminder of the subgoal to which it pertains.

4.6 Advancement or Solution: REWRITE_TAC

The function that implements HOL's rewriting scheme maps a list of theorems (to be used as left-to-right rewrite rules) to a tactic. For a given list `l`, the tactic `REWRITE_TAC l` (or any of the several variants of `REWRITE_TAC`, including `ASM_REWRITE_TAC` and so on – see ...) can produce a variable number of subgoals: either none or one. That is, a goal can be solved by rewriting, or it can be advanced to a single subgoal. In the former case, as for `ACCEPT_TAC`, an empty list of subgoals ensues. In the latter, the subgoal produced is unchanged as regards its assumption list, but may be changed as regards the term.

4.6.1 Solution by REWRITE_TAC

The following list, containing one pre-proved HOL theorem, can be used to complete the proof in the previous example (Section 4.5):

```
#let l = [ADD_CLAUSES];;
l =
[|- (0 + m = m) /\
    (m + 0 = m) /\
    ((SUC m) + n = SUC(m + n)) /\
    (m + (SUC n) = SUC(m + n))]
: thm list
```

In both the basis and step cases of that proof, it is sufficient to rewrite using `ADD_CLAUSES`, using any assumptions pertaining at the time of rewriting,

and using a standard list of basic tautologies[12]. This strategy is implemented by the tactic `ASM_REWRITE_TAC l`. Thus the goal is solved by the tactic

```
NAMED_INDUCT_TAC THEN
NAMED_ASM_REWRITE_TAC l
```

Once the corresponding named tactic `NAMED_REWRITE_TAC` is implemented, the procedure for printing the account of the rewriting proof step must choose between two ways of presenting the rewriting step: one which gives a wording appropriate to solution, and one for advancement only.

For solution, the account below shows the presentation of the (advancing) rewriting step in both cases:

```
This is the proof of the conjecture
>> example_8:
   "!m n p. m + (n + p) = (m + n) + p"

>>>> The proof is by mathematical induction on "m".
     This gives two cases to prove, the basis and step:

>> basis:
   "!n p. 0 + (n + p) = (0 + n) + p"

>> induction step:
   "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p"
   Assuming
    The induction hypothesis: "!n p. m + (n + p) = (m + n) + p"

The proof of the
>> basis:
   "!n p. 0 + (n + p) = (0 + n) + p"

is as follows:
>>>> This follows by using the equality,
     |- (0 + m = m) /\
        (m + 0 = m) /\
        ((SUC m) + n = SUC(m + n)) /\
        (m + (SUC n) = SUC(m + n))
     basic logical identities, and the assumptions made thus far.

This establishes

|- !n p. 0 + (n + p) = (0 + n) + p

The proof of the
>> induction step:
   "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p"
   Assuming
    The induction hypothesis: "!n p. m + (n + p) = (m + n) + p"
```

[12]All of HOL's rewriting functions use these basic rewrite rules except those with names suffixed by 'PURE', such as PURE_ASM_REWRITE_TAC.

```
is as follows:
>>>> This follows by using the equality,
     |- (0 + m = m) /\
        (m + 0 = m) /\
        ((SUC m) + n = SUC(m + n)) /\
        (m + (SUC n) = SUC(m + n))
     basic logical identities, and the assumptions made thus far.

This establishes

!n p. m + (n + p) = (m + n) + p
|- !n p. (SUC m) + (n + p) = ((SUC m) + n) + p

This establishes

|- !m n p. m + (n + p) = (m + n) + p

This completes the proof of the conjecture
>> example_8:
   "!m n p. m + (n + p) = (m + n) + p"
```

In the implementation of `NAMED_REWRITE_TAC`, the list provided of potential rewrite theorems is saved so that it can be printed as part of the account of the rewriting step. A more sophisticated account would perhaps not report every potential rewrite theorem, but only those on which changes to the term of the goal were based. Likewise, a more informative account would indicate, in both cases, *which*, if any, of the basic logical identities were engaged, and which, if any, of the assumptions – by name. However, to report only the rewrites actually engaged is beyond the scope of the current accounting method, which implements named tactics based on the values that would be produced by the corresponding ordinary tactics (on the corresponding ordinary goals). The method treats the ordinary tactics as 'black boxes'. To cause the ordinary rewriting tactic to keep a record of rewrites actually engaged would involve re-implementing the existing rewriting tactic (which happens to be particularly complex).

However, an analysis of the account shown, giving all *potential* rewrite rules, does have a use: an analysis of the account might suggest to the user some improvements to the tactic used. In the basis case, for example, the function `ASM_REWRITE_TAC` was specified, but in fact it is obvious from the account that no assumptions are present, and so `REWRITE_TAC` would have sufficed. The user could then decide whether

```
NAMED_INDUCT_TAC THENL
   [NAMED_REWRITE_TAC l;
    NAMED_ASM_REWRITE_TAC l]
```

were preferable to the original tactic.

It is worth noting here that the subgoal-theorem tree constructed in the process of accounting is structured exactly as the goal-oriented proof is actually performed. That is, although the original tactic is specified as a 'linear' sequence of two tactics, the induction proof step in fact yields two subgoals; the sequencing functional `THEN` is defined so as to apply its second argument to *all* the subgoals produced by its first argument. In this way, the account clarifies the proof's actual structure in a way that is not necessarily made apparent by the ML expression that generates the proof.

4.6.2 Advancement by REWRITE_TAC

The account of applying the following alternative tactic to the goal illustrates the wording for rewriting steps that do not solve goals. (It also happens to demonstrate the simpler tactic that is sufficient in the basis case.) It divides the rewriting step for the step case into two rewriting steps (the second using the basic rewrites and assumptions only), but is still a linear tactic.

```
NAMED_INDUCT_TAC THEN
NAMED_REWRITE_TAC[ADD_CLAUSES] THEN
NAMED_ASM_REWRITE_TAC[]
```

The account is then as follows, illustrating (in the step case) the wording for a rewriting step that does not solve a subgoal:

```
This is the proof of the conjecture
>> example_8:
   "!m n p. m + (n + p) = (m + n) + p"

>>>> The proof is by mathematical induction on "m".
     This gives two cases to prove, the basis and step:

>> basis:
   "!n p. 0 + (n + p) = (0 + n) + p"

>> induction step:
   "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p"
   Assuming
    The induction hypothesis: "!n p. m + (n + p) = (m + n) + p"

The proof of the
>> basis:
   "!n p. 0 + (n + p) = (0 + n) + p"

is as follows:
>>>> This follows by using the equality
     |- (0 + m = m) /\
```

```
        (m + 0 = m) /\
        ((SUC m) + n = SUC(m + n)) /\
        (m + (SUC n) = SUC(m + n))
     and basic logical identities.

This establishes

|- !n p. 0 + (n + p) = (0 + n) + p

The proof of the
>> induction step:
   "!n p. (SUC m) + (n + p) = ((SUC m) + n) + p"
   Assuming
    The induction hypothesis: "!n p. m + (n + p) = (m + n) + p"

is as follows:
>>>> Using the following equality
     |- (0 + m = m) /\
        (m + 0 = m) /\
        ((SUC m) + n = SUC(m + n)) /\
        (m + (SUC n) = SUC(m + n))
     and using basic tautologies, it is sufficient to prove:

>> "!n p. SUC(m + (n + p)) = SUC((m + n) + p)"
   Assuming
    The induction hypothesis: "!n p. m + (n + p) = (m + n) + p"

>>>> This follows from basic logical identities, as well as
     the assumptions made thus far.

This establishes

!n p. m + (n + p) = (m + n) + p
|- !n p. SUC(m + (n + p)) = SUC((m + n) + p)

This establishes

!n p. m + (n + p) = (m + n) + p
|- !n p. (SUC m) + (n + p) = ((SUC m) + n) + p

This establishes

|- !m n p. m + (n + p) = (m + n) + p

This completes the proof of the conjecture
>> example_8:
   "!m n p. m + (n + p) = (m + n) + p"
```

The account clarifies the fact that the first rewriting step of the linear tactic solves the basis case; this, again, is not immediately apparent from the ML procedure.

Finally, as the point about invalidity made in Section 4.4 applies also to the rewriting functions (which take a list of theorems as their parameter); any of the theorems on the list can introduce implicit assumptions, and these

assumptions are treated just as for substitution.[13]

4.7 Adding an Assumption: ASSUME_TAC

Like several tactics so far, `ASSUME_TAC` maps a theorem to a tactic. It simply adds the conclusion of the theorem provided to the assumption list, and justifies this step by discharging the assumption and applying *Modus Ponens*. Thus, an implicit assumption may again be introduced. The following account shows the wording for printing such a proof step, and illustrates how implicit assumptions can be raised by using `ASSUME_TAC` to access assumptions by text (a common method in HOL proofs).

For example, suppose we wish to prove

```
mk_named_goal(‘example_9‘, [], "(p = q) ==> (q = r) ==> (p = r)")
```

(for `p`, `q` and `r`) of some given type, by assuming the antecedents in turn, appealing to the transitivity of equality to derive as a new assumption `"p = r"`, and then using the new assumption as a rewrite rule. The account of this proof, using the corresponding `NAMED_ASSUME_TAC` is:

```
This is the proof of the conjecture
>> example_9:
   "(p = q) ==> (q = r) ==> (p = r)"

>>>> It is sufficient to prove:

>> "(q = r) ==> (p = r)"
   Assuming
    The antecedent: "p = q"

>>>> It is sufficient to prove:

>> "p = r"
   Assuming
    The antecedent: "q = r"
    The antecedent: "p = q"

>>>> We use the fact that
     p = q, q = r |- p = r.
     It is sufficient to prove:

>> "p = r"
```

[13] Indeed, any of the assumptions that happens to be engaged as a rewrite rule by ASM_REWRITE_TAC – but not those which are not – must also, necessarily, introduce an implicit assumption. However, these particular implicit assumptions seem unlikely to cause confusion, and so are not recorded as implicit in the accounts.

```
   Assuming
    The added hypothesis: "p = r"
    The antecedent: "q = r"
    The antecedent: "p = q"
   Assuming implicitly
    The hypothesis of the theorem used: "p = q"
    The hypothesis of the theorem used: "q = r"
>>>> This follows from basic logical identities, as well as
     the assumptions made thus far.
This establishes
p = r |- p = r
This establishes
p = q, q = r |- p = r
This establishes
p = q |- (q = r) ==> (p = r)
This establishes
|- (p = q) ==> (q = r) ==> (p = r)
This completes the proof of the conjecture
>> example_9:
   "(p = q) ==> (q = r) ==> (p = r)"
```

Note that the second theorem established (`p = q, q = r |- p = r`) carries as hypotheses the two implicit assumptions of the theorem parameter to `ASSUME_TAC`. This assumption step is explained by the phrase '`We use the fact that` ...'.

5 Conversions

A conversion in HOL is a function mapping a term to a theorem – that is, a theorem parameterized on a term. For example, the concept of beta-conversion is represented in HOL by the function `BETA_CONV` which maps a term (the beta redex) to a theorem expressing the reduction:

```
#BETA_CONV "(\x. x > 0) 3";;
|- (\x. x > 0)3 = 3 > 0
```

Conversions provide a way of deriving particular instances of facts which cannot themselves be expressed as theorems in the HOL logic. (To express

beta-conversion in general, for example, would require quantification over syntax classes of logical expressions.)

To enable the use of such equational theorems as reduction tactics, a function `CONV_TAC` is provided. `CONV_TAC` maps a given conversion to a tactic which will perform the reduction on a goal with suitable term. The tactic thus produced, when applied to a goal, will either fail to be applicable, or will produce exactly one subgoal.

```
#CONV_TAC BETA_CONV;;
- : tactic

#CONV_TAC BETA_CONV ([],"(\x. x > 0) 3");;
([([], "3 > 0")], -) : subgoals
```

The reduction is justified by a simple substitution.

To construct the account of a proof step generated by applying a tactic of the form `CONV_TAC` c, for some conversion c, the usual method is used. The theorem –

```
|- (\x. x > 0)3 = 3 > 0
```

– which justifies the beta-reduction step is saved as a theorem parameter in the account, for purposes of explanation. For example, to explain the application of the named tactic

```
NAMED_CONV_TAC BETA_CONV
```

to the goal

```
mk_named_goal(`example`,[],"(\x. x > 0) 3")
```

the account produced is:

```
This is the proof of the conjecture
>> example:
   "(\x. x > 0)3"

>>>> We use the instantiated theorem-schema
     |- (\x. x > 0)3 = 3 > 0
     making it sufficient to prove:

>> "3 > 0"

...

This establishes
```

```
|- 3 > 0
```

```
This establishes
```

```
|- (\x. x > 0)3
```

```
This completes the proof of the conjecture
>> example:
   "(\x. x > 0)3"
```

Because the named tactic records the particular fact that was used, the method gives a meaninful explanation however the conversion is expressed. For example, the function `DEPTH_CONV` is one of several functions which transform conversions to new conversions. The conversion (`DEPTH_CONV BETA_CONV` produces a conversion which will apply recursively – to arbitrary depth – to all the beta-redexes of a term.)
For example, to explain the application of the named tactic

```
NAMED_CONV_TAC (DEPTH_CONV BETA_CONV)
```

to the goal

```
(mk_named_goal(‘example‘,[],"(\x. x > 0)3 = ((\x. x > 0)4 = T)"))
```

the account produced is:

```
This is the proof of the conjecture
>> example:
   "(\x. x > 0)3 = ((\x. x > 0)4 = T)"
```

```
>>>> We use the instantiated theorem-schema
     |- ((\x. x > 0)3 = ((\x. x > 0)4 = T)) = (3 > 0 = (4 > 0 = T))
     making it sufficient to prove:
```

```
>> "3 > 0 = (4 > 0 = T)"
```

```
...
```

```
This establishes
```

```
|- 3 > 0 = (4 > 0 = T)
```

```
This establishes
```

```
|- (\x. x > 0)3 = ((\x. x > 0)4 = T)
```

```
This completes the proof of the conjecture
>> example:
   "(\x. x > 0)3 = ((\x. x > 0)4 = T)"
```

For complex expressions denoting a conversion, it could be quite difficult to reconstruct the tactic produced by `CONV_TAC` when applied to that conversion. The explanation makes it unnecessary to remember what form of theorem each conversion (such as `BETA_CONV`) gives on appropriate terms; what effects the various conversion transformers (such as `DEPTH_CONV`) have on conversions in general; and in what sense the parameterized tactic `CONV_TAC` produces a tactic given a conversion. The explanation instead supplies the actual equational theorem justifying the reduction.

6 Resolution

The 'resolution' tactics provided in HOL – `IMP_RES_TAC` and `RES_TAC` – are the basis of the second group of named tactics. Members of this group rely on the results of the corresponding ordinary tactics, but they further process the results so that they can be presented in a meaningful way[14].

The function `IMP_RES_TAC` maps a theorem to a tactic. It gives a way of bringing to bear an implicative[15] axiom or previously proved theorem on a goal by adding to the current assumptions of a goal a certain subset of the collective direct and indirect consequences of that theorem together with the current assumptions.

The consequences are found by attempting to match the antecedent of the implicative theorem to each existing assumption (i.e. candidate antecedent); so determining an instantiation, wherever a match is made. The appropriate instance of the consequent of the implication is then added as a new assumption, to the subgoal. A single application of `IMP_RES_TAC` *th* to a goal, for a theorem *th*, is sufficent for finding all new assumptions of the form sought; subsequent applications of `IMP_RES_TAC` have no further effect.

The instantiated consequents are processed before new the new subgoal(s) are constructed. If the consequent is an n-ary disjunction, n subgoals are created, one with each respective disjunct as a new assumption. If it is an n-ary conjunction, the n conjuncts are added separately to the (single) subgoal. (Existential and other consequents are not further processed.)

[14] The resolution tactics are mis-named in that they do not do resolution in the classical sense (based on unification), but simply some one-way matching of an implication to a candidate antecedent, followed by forward inference based on *Modus Ponens*.

[15] Implications, in this context, are taken in a generalized sense, as described in ...

The tactic RES_TAC, on a goal, looks for pairs of resolvents *within* the set of current assumptions. It considers each implicative assumption against the set of all other assumptions in the same way that IMP_RES_TAC resolves an implication against a set of assumptions. For each implication matched, RES_TAC similarly adds as a new assumption the appropriate instance of the consequent. Like IMP_RES_TAC, RES_TAC applied to a goal produces n subgoals when the consequent of a matched implication is an n-ry disjunction. The full set of results that RES_TAC is able to find is not necessarily found in a single application of the tactic; whether it is depends on the ordering of the initial assumptions.

Both IMP_RES_TAC *th* and RES_TAC can either solve goals or advance them. They can solve a goal either by deriving as a new assumption the term itself of the goal, or by deriving falsity as a new assumption (in which case *anything* desired could be established, including the particular term of the goal). Where these tactics advance goals, they can produce an indefinite number of subgoals; just one subgoal if no match made involves an implication with a disjunctive consequent; and more than one subgoal if at least one match does so. Where these tactics advance a goal, they can add to the assumption list, but they cannot change the term.

To construct the account of a proof step involving one of the resolution tactics involves computing the results of the corresponding ordinary tactic on the corresponding ordinary goal, then identifying the nature of the result, and (in the advancement case) naming the relevant parts of subgoals. Where the step solves the goal, direct solution and solution by contradiction are distinguished. This is done by checking whether an arbitrary goal would also be solved at that point. An appropriate string is then chosen to denote the proof step so that the two solution cases can be printed appropriately. For example, consider the pre-proved theorem LESS_MONO:

```
#LESS_MONO;;
|- !m n. m < n ==> (SUC m) < (SUC n)
```

In the following proof, LESS_MONO is used to solve a trivially easy goal by resolution:

```
#let g = ["p < q"],"SUC p < SUC q";;
g = (["p < q"], "(SUC p) < (SUC q)") : goal

#let gl,p = IMP_RES_TAC LESS_MONO g;;
gl = [] : goal list
```

```
p = - : proof

#let th = p[];;
th = . |- (SUC p) < (SUC q)

#print_all_thm th;;
p < q |- (SUC p) < (SUC q)
```

The account generated by the named tactic shows the wording used:

```
This is the proof of the conjecture
>> example:
   "(SUC p) < (SUC q)"
   Assuming
    The fact: "p < q"

>>>> This follows directly
     by using the assumptions made thus far and the fact
     |- !m n. m < n ==> (SUC m) < (SUC n).

This establishes

p < q |- (SUC p) < (SUC q)

This completes the proof of the conjecture
>> example:
   "(SUC p) < (SUC q)"
   Assuming
    The fact: "p < q"
```

The next example demonstrates solution by contradiction. (Since the term of the goal does not matter, we use an arbitrary provable term `t`.) The pre-proved theorem `LESS_NOT_EQ` is the theorem parameter:

```
#LESS_NOT_EQ;;
|- !m n. m < n ==> ~(m = n)
```

The implication of `LESS_NOT_EQ` is taken by `IMP_RES_TAC` to be a form of the canonical

```
|- m < n ==> (m = n) ==> F
```

Though `IMP_RES_TAC` this time succeeds by deriving a contradiction, there is nothing in the following ordinary HOL session to indicate that fact:

```
#let g = ["p < q";"(p:num) = q"],"t:bool";;
g = (["p < q"; "p = q"], "t") : goal

#let gl,p = IMP_RES_TAC LESS_NOT_EQ g;;
gl = [] : goal list
```

```
p = - : proof

#let th = p[];;
th = .. |- t

#print_all_thm th;;
p < q, p = q |- t
```

The wording of the account makes the proof method clear[16]:

```
This is the proof of the conjecture
>> example:
   "t"
   Assuming
    The fact1: "p < q"
    The fact2: "p = q"

>>>> This follows by contradiction,
     using the assumptions made thus far and the fact
     |- !m n. m < n ==> ~(m = n).

This establishes

p < q, p = q |- t

This completes the proof of the conjecture
>> example:
   "t"
   Assuming
    The fact1: "p < q"
    The fact2: "p = q"
```

Where resolution advances a goal rather than solving it, this is indicated in the account; the new result is identified. Here, `LESS_MONO` is again used:

```
This is the proof of the conjecture
>> example:
   "t"
   Assuming
    The fact: "p < q"

>>>> From the assumptions made thus far and the fact
     |- !m n. m < n ==> (SUC m) < (SUC n),
     it is sufficient to prove the following:

>> "t"
   Assuming
    The consequence: "(SUC p) < (SUC q)"
    The fact: "p < q"

...
```

[16] An alternative presentation could print the canonical form of LESS_NOT_EQ (i.e. the form actually used by the tactic), if that were felt to be more informative.

```
This establishes

(SUC p) < (SUC q), p < q |- t

This establishes

p < q |- t

This completes the proof of the conjecture
>> example:
   "t"
   Assuming
    The fact: "p < q"
```

Of course, there may be more than one new result; in that case, the new results are numbered in the order in which they would ordinarily be added to the assumptions in HOL:

```
This is the proof of the conjecture
>> example:
   "t"
   Assuming
    The fact1: "p1 < q1"
    The fact2: "p2 < q2"

>>>> From the assumptions made thus far and the fact
     |- !m n. m < n ==> (SUC m) < (SUC n),
     it is sufficient to prove the following:

>> "t"
   Assuming
    The consequence 2: "(SUC p2) < (SUC q2)"
    The consequence 1: "(SUC p1) < (SUC q1)"
    The fact1: "p1 < q1"
    The fact2: "p2 < q2"

....

This establishes

(SUC p2) < (SUC q2), (SUC p1) < (SUC q1), p1 < q1, p2 < q2 |- t

This establishes

p1 < q1, p2 < q2 |- t

This completes the proof of the conjecture
>> example:
   "t"
   Assuming
    The fact1: "p1 < q1"
    The fact2: "p2 < q2"
```

As noted earlier, a resolvent with a disjunctive conclusion can cause a case split. If that happens, the cases are numbered and identified in the

account (and new results identified as before). In the following example, we resolve against the pre-proved `LESS_LEMMA`:

```
#let LESS_LEMMA1 = theorem `prim_rec` `LESS_LEMMA1`;;
LESS_LEMMA1 = |- !m n. m < (SUC n) ==> (m = n) \/ m < n

This is the proof of the conjecture
>> example:
   "t"
   Assuming
    The fact: "p < (SUC q)"

>>>> From the assumptions made thus far and the fact
     |- !m n. m < (SUC n) ==> (m = n) \/ m < n,
     it is sufficient to prove the following:

>> disjunctive case 1 of 2:
   "t"
   Assuming
    The consequence: "p = q"
    The fact: "p < (SUC q)"

>> disjunctive case 2 of 2:
   "t"
   Assuming
    The consequence: "p < q"
    The fact: "p < (SUC q)"

The proof of the
>> disjunctive case 1 of 2:
   "t"
   Assuming
    The consequence: "p = q"
    The fact: "p < (SUC q)"

is as follows:

....

This establishes

p = q, p < (SUC q) |- t

The proof of the
>> disjunctive case 2 of 2:
   "t"
   Assuming
    The consequence: "p < q"
    The fact: "p < (SUC q)"

is as follows:

....

This establishes

p < q, p < (SUC q) |- t
```

```
This establishes
p < (SUC q) |- t
This completes the proof of the conjecture
>> example:
   "t"
   Assuming
    The fact: "p < (SUC q)"
```

Finally, the point made in (...) about implicit assumptions applies to any tactic of the form `IMP_RES_TAC` *th*; implicit assumptions may be introduced by the theorem parameter *th*. `RES_TAC` does not have this property.

The implementations of `NAMED_IMP_RES_TAC` and `NAMED_RES_TAC` follow the outlines of simpler implementations (...) but involve rather more processing of the ordinary results in order to build useful accounts into the named functions.

7 Popping Assumptions

There are several groups of functions in HOL whose members produce new tactics from old. Such functions might be called 'tactic transformers'. One such group contains the HOL function `POP_ASSUM`, which maps a function `f` of type `thm -> tactic` to a new function of type `tactic` so that

```
POP_ASSUM f = \((a.A),t). f (ASSUME a) (A,t)
```

That is, the function `POP_ASSUM` transforms `f` into a tactic which takes a goal (with at least one term on the assumption list), removes the *first* term (`a`) on the assumption list, assumes that term (to produce the theorem `a |- a`), supplies that theorem to the function `f` (to yield a new tactic), and finally, applies that tactic to the reduced goal (the goal without the leading assumption).

The other two members of this group of functions are `POP_ASSUM_LIST` and `SUBST_ALL_TAC`. The method of viewing the assumption list of a goal as a stack which can be 'popped' was developed for LCF by Larry Paulson (...).

The reasons for wishing to pop or remove an assumption before using it may not be immediately apparent, as this technique does not correspond to any natural strategy. For example, in the textbook proof shown in (...), one of the proof steps was:

> If n itself is a prime, there is nothing to prove. Suppose, then, that n is composite...

The argument then continues until the desired fact is established for n, under the assumption that n is composite; and the assumption is used at some point. It would sound very odd if, after the assumption were used, but before the case were solved, the proof were to continue:

> ...We now cease to assume that n is composite, as this fact is no longer required.

This sounds odd because assumptions in a normal subgoaling framework cannot be 'dropped' once they have been used, and they would normally *be* used once introduced. In the example, the assumption that n is composite persists from subgoal to subgoal, past the point of its use, right until the composite case of the proof is established. However, in proofs in HOL, there are at least two reasons for wishing to give the appearance of dropping an assumption from a subgoal, and one reason for actually doing so.

7.1 Popping to Erase Used Assumptions

The simplest reason for causing an assumption to seem to vanish is that during an interactive session in which proof steps are made one at a time, each subgoal of the proof tree is printed out to the user explicitly. To reduce apparent clutter, it has become a common practice to use the function `POP_ASSUM` to supress the printing of assumptions that were but are no longer required. Thus, application of the tactic `POP_ASSUM SUBST1_TAC` not only effects a substitution (and without explicit mention of the substitution equation – i.e. of the leading assumption), but also prevents the leading assumption from appearing subsequently in the subgoal. It does not, of course, prevent the theorem achieving the original goal from depending on the popped assumption, since the justification of `POP_ASSUM SUBST1_TAC` necessarily adds the popped assumption to any theorem achieving the subgoal.

```
#let g = ["x = 5"],"x > 0";;
g = (["x = 5"], "x > 0") : goal
#let gl,p = POP_ASSUM SUBST1_TAC g;;
gl = [([], "5 > 0")] : goal list
```

```
p = - : proof

...

th = |- 5 > 0

#print_all_thm(p[th]);;
x = 5 |- x > 0

...

th' = x = 5 |- 5 > 0

#print_all_thm(p[th']);;
x = 5 |- x > 0
```

7.2 Popping to Replace an Assumption

The second reason for popping an assumption is to re-introduce it immediately in a different form. For example, it may be convenient to 'replace' an assumption of the form $t1 = t2$ with the equivalent $t2 = t1$, in which case the original assumption is no longer required, and indeed, may be an obstacle if it does not co-exist happily with the new form (in this case, for example, it would prevent a subsequent application of `ASM_REWRITE_TAC...` from terminating). One way to achieve this is illustrated below:

```
#let g = ["5 = x"],"t:bool";;
g = (["5 = x"], "t") : goal

#let gl,p = POP_ASSUM (ASSUME_TAC o SYM) g;;
gl = [(["x = 5"], "t")] : goal list
p = - : proof
```

Again, the justification of `POP_ASSUM (ASSUME_TAC o SYM)` necessarily produces a theorem depending on the popped assumption `5 = x`, given a theorem achieving the subgoal – so the popped theorem is not gone, but simply not printed.

7.3 Popping to Erase Irrelevant Assumptions

The third reason for popping assumptions is that in HOL proofs in which certain kinds automation come into play, useless assumptions *are* sometimes introduced into subgoals; the resolution tactics (...), which add to the assumptions of a goal all the collective consequences of a certain type of the existing assumptions (with or without an additional implicative lemma), are

notorious for this. Useless assumptions are therefore popped (and genuinely dropped) in order to reduce the confusion (and clutter) that might result from the presence of assumptions which are never used and on which nothing ever actually depends. For example:

```
#let g = ["5 = x"],"t:bool";;
g = (["5 = x"], "t") : goal

#let gl,p = POP_ASSUM (\th. ALL_TAC) g;;
gl = [([], "t")] : goal list
p = - : (* list -> *)

...

th = |- t

#p[th];;
|- t
```

In this case, the assumption `5 = x` is genuinely lost; the justification of `POP_ASSUM (\th. ALL_TAC)`[17] – or, to use a combinator, `POP_ASSUM (K ALL_TAC)` – does not add the popped assumption to the theorem achieving the goal. This cases arises for any user-defined function which shares the property of genuinely dropping assumptions.

It is also the case that if the achieving theorem does depend on the lost assumption, the justification still maps that theorem to a theorem achieving the original goal, even though the subgoal is not achieved:

```
#let g = ["5 = x"],"t:bool";;
g = (["5 = x"], "t") : goal

#let gl,p = POP_ASSUM (K ALL_TAC) g;;
gl = [([], "t")] : goal list
p = - : (* list -> *)

...

#print_all_thm th;;
5 = x |- t

#print_all_thm(p[th]);;
5 = x |- t
```

7.4 Accounting for Popping Assumptions

For whatever reasons it is used, the assumption-popping strategy is perfectly valid, since a theorem that achieves the subgoal less an assumption must

[17] as POP_ASSUM is currently implemented in HOL

also achieve a subgoal *with* that assumption, by the definition of achievement. Whether, in each case, popping assumptions is the best method for producing the desired effect is a question of style, taste and clarity, but this is not the question of interest here. Instead, the question is how to produce a natural account of a proof that relies on this technical and non-natural device.

The key to producing such accounts, in the first and second cases, is the concept of an *implicit assumption*, introduced in (...). This is suggested by the way assumptions not visible in subgoals are nevertheless known to justifications, exactly as happens when a tactic is applied which has been constructed from a theorem whose hypotheses do not correspond to current assumptions.

The account desired would simply document the tactic actually applied, show the subgoal with the popped assumption no longer explicit, but leave no mystery about the persistence of the assumption in the justification. That is, the popped assumption would appear as implicit where it ceased to appear as explicit.

7.4.1 Accounting for Popping to Erase Used Assumptions

A sensible account of the first case (popping to erase used assumptions) is constructed by first defining a function `NAMED_POP_ASSUM` in parallel with HOL's `POP_ASSUM` function. Thus, for a function `f` of type `thm -> named_tactic`, the function `NAMED_POP_ASSUM f` is a named tactic which when applied to a named goal

1. finds the term part (`tm`, say) of the first *explicit* assumption of a goal;
2. assumes `tm` to give a theorem `tm |- tm` and applies `f` to the resulting theorem to form a named tactic; and
3. applies the named tactic `f(ASSUME tm)` to the named goal *minus* its first explicit assumption.

This means that in relation to the reduced goal, the new tactic is bringing to bear a theorem which depends on a hypothesis not represented in the goal – namely, `tm`. Thus the situation is the same as in (...). The account produced for the first case(Section 7.1), in which the tactic `POP_ASSUM SUBST1_TAC` was used to substitute with *and* dispense with the leading assumption, is as follows:

```
This is the proof of the conjecture
>> example1:
   "x > 0"
   Assuming
    The fact: "x = 5"

>>>> We substitute according to the following equality:
     x = 5 |- x = 5.
     Thus, it is sufficient to prove:

>> "5 > 0"
   Assuming implicitly
    The hypothesis of the equality: "x = 5"

...

This establishes

|- 5 > 0

This establishes

x = 5 |- x > 0

This completes the proof of the conjecture
>> example1:
   "x > 0"
   Assuming
    The fact: "x = 5"
```

This interpretation of `NAMED_POP_ASSUM` assures that when the popped assumption (`tm`) is actually used (e.g. in this case, by the substitution tactic `NAMED_SUBST1_TAC(ASSUME tm)`), it will necessarily be recorded in the substitution subgoal as an implicit assumption. The account describes just one proof step: the substitution. It does not mention the popping function, but simply documents the 'loss' of the explicit assumption at the point of substitution, where the implicit assumption arises. This gives the effect of transferring the popped, explicit assumption to the list of implicit assumptions, which is what was desired.

A different interpretation of `NAMED_POP_ASSUM f` is to insist that a popped assumption *always* be recorded as implicit. To implement this view, the goal to which the tactic `f(ASSUME tm)` is applied does not have the popped assumption removed, but simply marked as implicit.

If an implicit assumption is ultimately recorded in the first way, then the same assumption is recorded as implicit in the new way. However, the advantage of the new method over the first is that the new method is not committed to the phrasing with which, in the first way, the function `f` identifies the implicit assumption – indicating that the assumption was used invalidly.

The first method *is* committed to this phrasing, as it is built into the account produced by the justification of `f`; the name of the assumption before it was popped cannot be restored. (In the example, the implicit assumption is labelled `The hypothesis of the equality` by `NAMED_SUBST1_TAC`.) Using the new method, the name borne by the assumption in the previous subgoal (`fact`, in the example) could be retained (or some other preferred phrase used instead). The disadvantage of the new method is that it does not cover the third case (popping to erase irrelevant assumptions); we return to this point in Section 7.4.3.

An elaboration of `NAMED_POP_ASSUM f` is to have it notice when `f` is exactly equivalent to `NAMED_ASSUME_TAC`, in which case there is no overall effect. In that case, the justification of `f` can be replaced with the identity justification (i.e. the function mapping a list containing one account to that account) so that instead of the account

```
This is the proof of the conjecture
>> example1:
   "x > 0"
   Assuming
    The fact: "x = 5"

>>>> We use the assumption that
     x = 5 |- x = 5.
     It is sufficient to prove:

>> "x > 0"
   Assuming
    The added hypothesis: "x = 5"
   Assuming implicitly
    The hypothesis of the theorem used: "x = 5"

...

This establishes

x = 5 |- x > 0

...
```

which documents the *double* re-assumption of `x = 5` without it ever obviously having been lost, the following less confusing account is produced:

```
This is the proof of the conjecture
>> example1:
   "x > 0"
   Assuming
    The fact: "x = 5"

...
```

```
This establishes
x = 5 |- x > 0
...
```

This is a minor elaboration, as the exact situation described is infrequent, and the trick does not extend to anything more complex (i.e. to anything involving modification of the justification of `f`).

7.4.2 Accounting for Popping to Replace Assumptions

The original interpretation of popping also gives a reasonable account of the second case: popping to replace an assumption (Section 7.2). In the example used, the tactic `POP_ASSUM (\th. ASSUME_TAC(SYM th))` was used to drop an old assumption and add a new one, as if replacing the old one. The account produced is:

```
This is the proof of the conjecture
>> example2:
   "t"
   Assuming
    The fact: "5 = x"

>>>> We use the fact that
     5 = x |- x = 5.
     It is sufficient to prove:

>> "t"
   Assuming
    The added hypothesis: "x = 5"
   Assuming implicitly
    The hypothesis of the theorem used: "5 = x"

...
This establishes
x = 5 |- t
This establishes
5 = x |- t
This completes the proof of the conjecture
>> example2:
   "t"
   Assuming
    The fact: "5 = x"
```

Again, by using the new interpretation of popping (i.e. by insisting that popped assumptions are immediately made implicit) the phrase

```
The hypothesis of the theorem used
```

identifying the implicit assumption, could be varied as desired and does not have to be the one seen above, which was supplied by `ASSUME_TAC`.

7.4.3 Accounting for Popping to Erase Irrelevant Assumptions

The original interpretation of `NAMED_POP_ASSUM` also gives a natural account of the third case (Section 7.3), in which an unnecessary assumption is actually dropped, and is *not* stitched into any justification function. In the example shown, `POP_ASSUM (\th. ALL_TAC)` (i.e. `POP_ASSUM (K ALL_TAC)`) was used to give this effect. The account produced is:

```
This is the proof of the conjecture
>> example3:
   "t"
   Assuming
    The fact: "5 = x"

>>>> It is sufficient to prove:

>> "t"

...

This establishes

|- t

This establishes

|- t

This completes the proof of the conjecture
>> example3:
   "t"
   Assuming
    The fact: "5 = x"
```

The account documents the loss of the assumption (a valid step), and shows that when the subgoal is ultimately achieved, the justification of the proof step returns a theorem which does *not* depend on the original (and lost) fact. This corresponds to – and explains – the behaviour of `POP_ASSUM (K ALL_TAC)` in HOL, as shown in Section 7.3.

It is also the case, as mentioned in Section 7.3, that the justification of `POP_ASSUM (K ALL_TAC)` maps the theorem `5 = x |- t` to itself, and so achieves the original goal, even though the theorem does not achieve the

subgoal. If the theorem `5 = x |- t` is eventually established and then supplied as the purported achievement of the subgoal, the following account results:

```
This is the proof of the conjecture
>> example3:
   "t"
   Assuming
    The fact: "5 = x"

>>>> It is sufficient to prove:

>> "t"

...

This establishes

5 = x |- t

which does not satisfy
>> "t"

This establishes

5 = x |- t

This completes the proof of the conjecture
>> example3:
   "t"
   Assuming
    The fact: "5 = x"
```

The local failure is noted, as well as the ultimate achievement of the original goal. This also corresponds to – and explains – the behaviour of `POP_ASSUM (K ALL_TAC)` in HOL.

In contrast to this interpretation of `NAMED_POP_ASSUM f` – in which the popped assumption (`tm`) is allowed to appear or not in the course of applying the tactic `f (ASSUME tm)` to the goal containing no version of the assumption – is the second interpretation, in which the popped assumption is made implicit in the goal to which the tactic is applied. (We call this function `NAMED_POP_TRACE` because it necessarily leaves a 'trace' of the popped assumption.) Under the second interpretation, the account is:

```
This is the proof of the conjecture
>> example3:
   "t"
   Assuming
    The fact: "5 = x"

>>>> It is sufficient to prove:
```

```
>> "t"
   Assuming implicitly
    The fact: "5 = x"

...
This establishes

|- t

This establishes

|- t

This completes the proof of the conjecture
>> example3:
   "t"
   Assuming
    The fact: "5 = x"
```

Here, a record of the popped assumption is kept, so it is not definitively lost. This still corresponds to HOL's behaviour, but it no longer satisfies the original definition of implicit assumptions, which was based on the behaviour of justifications. It seems desirable to retain the present definition of implicit assumptions as the basis for explaining why certain assumptions do or do not appear as hypotheses of certain theorems. Therefore, it seems sensible to retain the original view of the pop operation, which covers all three cases adequately. However, there is another use for this version of the pop function; it arises in the next section.

An alternative might be to implement `NAMED_POP_ASSUM f` differently for different `f`, using the original view of popping for cases resembling the third case and the new view in others. Probably, the choice would have to be represented by a conditional within the implementation of a more general pop function, as there seems no way in advance to tell which sort of function `f` one has been given. This would be a complicated way around the problem, if it could be made to work at all.

The root of the difficult with `K ALL_TAC` is the definition of achievement in HOL. This specifies that a theorem's hypotheses need only be a subset of the assumptions of the subgoal it purports to achieve. If the definition required the full set, the problem would not arise. A less drastic modification of HOL, however, would at least produce uniformity over all functions to which the pop operator could be applied; that would be to re-implement HOL's function `POP_ASSUM` so that for any appropriate `f`, the justification of `POP_ASSUM f` were not simply the justification (`p`, say) of `f`, but rather `(ADD_ASSUM tm) o p`.

where `ADD_ASSUM : term -> thm -> thm` is the inference rule in HOL that adds a hypothesis to a theorem. Under this definition, the two views of the pop operator would be the same, so we could use the second, if desired, to choose a way of identifying the popped assumption.

7.5 Accounting for POP_ASSUM_LIST

The function `POP_ASSUM_LIST` is a generalization of `POP_ASSUM` which removes all of the assumptions of a goal and passes the list of (assumed) assumptions to a function of type `thm list -> tactic`. The account is therefore similar; for example, the following proof

```
#let g = ["x = 5";"y = 4"],"x > y";;
g = (["x = 5"; "y = 4"], "x > y") : goal

#let gl,p = POP_ASSUM_LIST SUBST_TAC g;;
gl = [([], "5 > 4")] : goal list
p = - : proof

...

th = |- 5 > 4

#print_all_thm(p[th]);;
x = 5, y = 4 |- x > y
```

receives the following account:

```
This is the proof of the conjecture
>> example4:
   "x > y"
   Assuming
    The fact: "x = 5"
    The fact: "y = 4"

>>>> We substitute according to the following equalities:
     x = 5 |- x = 5
     y = 4 |- y = 4.
     Thus, it is sufficient to prove:

>> "5 > 4"
   Assuming implicitly
    The hypothesis of the equality: "x = 5"
    The hypothesis of the equality: "y = 4"

...

This establishes

|- 5 > 4
```

```
This establishes

x = 5, y = 4 |- x > y

This completes the proof of the conjecture
>> example4:
   "x > y"
   Assuming
    The fact: "x = 5"
    The fact: "y = 4"
```

7.6 Accounting for SUBST_ALL_TAC

The function `SUBST_ALL_TAC`, of type `thm -> tactic`, is not a tactic transformer, but the tactic `SUBST_ALL_TAC` *th* shares with `POP_ASSUM` *f* the property of causing assumptions of a goal to seem to disappear. `SUBST_ALL_TAC` uses an equational theorem to effect a substitution throughout the term of the goal – in the style of `SUBST1_TAC` – and *also* to effect the substitution throughout the assumption list. In particular, `SUBST_ALL_TAC` *th* resembles `POP_ASSUM` *f* when the latter is used for replacing assumptions by equivalent terms – and at the same time, making the original assumptions implicit (Section 7.2). This, again, does not correspond to a natural pattern of reasoning, and that makes it difficult to give a natural account. The effects of `SUBST_ALL_TAC` are illustrated in the following example:

```
rth = |- x = 1

#let g = ["(y:num) = x";"w > x";"w < 5"],"(z:num) = x";;
g = (["y = x"; "w > x"; "w < 5"], "z = x") : goal

#let gl,p = SUBST_ALL_TAC rth g;;
gl = [(["y = 1"; "w > 1"; "w < 5"], "z = 1")] : goal list
p = - : proof

...

th = y = 1, w > 1, w < 5 |- z = 1

#print_all_thm(p[th]);;
y = x, w > x, w < 5 |- z = x
```

In HOL, `SUBST_ALL_TAC` happens to be implemented as an application of `SUBST1_TAC` (to modify the term of the goal), sequenced with a application of `POP_ASSUM_LIST` (Section 7.5) to a function that substitutes through and re-assumes each assumption (to modify the assumptions). That is, to modify

the assumptions, all are removed, and each is transformed, then re-assumed. Although the method of implemening named tactics so far has not been to parallel the actual HOL implementation – the HOL functions are taken as 'black boxes' – one reason for trying to do so in this case is to leave the recording of the implicit assumptions to the `ASSUME_TAC`'s, so that it is automatic.

The parallel implemantation satisfies:

```
NAMED_SUBST_ALL_TAC rth =
NAMED_SUBST1_TAC rth THEN
NAMED_POP_ASSUM_LIST
   (\[th1;...;thn]. ASSUME_TAC (SUBS [rth] thn)
                    THEN
                    .
                    .
                    .
                    THEN
                    ASSUME_TAC (SUBS [rth] th1))
```

The account that is produced in this way turns out to be rather inscrutable. Although this implementation of `NAMED_SUBST_ALL_TAC` gives the correct end result, the intermediate proof steps – normally not visible – are not what one would expect; they reveal local failures of theorems to achieve subgoals:

```
This is the proof of the conjecture
>> example5:
   "z = x"
   Assuming
    The fact1: "y = x"
    The fact2: "w > x"
    The fact3: "w < 5"

>>>> We substitute according to the following equality:
     |- x = 1.
     Thus, it is sufficient to prove:

>> "z = 1"
   Assuming
    The fact1: "y = x"
    The fact2: "w > x"
    The fact3: "w < 5"

>>>> We use the assumption that
     w < 5 |- w < 5.
     It is sufficient to prove:

>> "z = 1"
   Assuming
    The added hypothesis: "w < 5"
   Assuming implicitly
    The hypothesis of the theorem used: "w < 5"
```

```
>>>> We use the fact that
     w > x |- w > 1.
     It is sufficient to prove:

>> "z = 1"
   Assuming
    The added hypothesis: "w > 1"
    The added hypothesis: "w < 5"
   Assuming implicitly
    The hypothesis of the theorem used: "w > x"
    The hypothesis of the theorem used: "w < 5"

>>>> We use the fact that
     y = x |- y = 1.
     It is sufficient to prove:

>> "z = 1"
   Assuming
    The added hypothesis: "y = 1"
    The added hypothesis: "w > 1"
    The added hypothesis: "w < 5"
   Assuming implicitly
    The hypothesis of the theorem used: "y = x"
    The hypothesis of the theorem used: "w > x"
    The hypothesis of the theorem used: "w < 5"

...

This establishes

y = 1, w > 1, w < 5 |- z = 1

This establishes

w > 1, w < 5, y = x |- z = 1

which does not satisfy
>> "z = 1"
   Assuming
    The added hypothesis: "w > 1"
    The added hypothesis: "w < 5"
   Assuming implicitly
    The hypothesis of the theorem used: "w > x"
    The hypothesis of the theorem used: "w < 5"

This establishes

w < 5, y = x, w > x |- z = 1

which does not satisfy
>> "z = 1"
   Assuming
    The added hypothesis: "w < 5"
   Assuming implicitly
    The hypothesis of the theorem used: "w < 5"

This establishes

y = x, w > x, w < 5 |- z = 1
```

```
This establishes

y = x, w > x, w < 5 |- z = x

This completes the proof of the conjecture
>> example5:
   "z = x"
   Assuming
    The fact1: "y = x"
    The fact2: "w > x"
    The fact3: "w < 5"
```

That is, the first subgoal correctly reflects the modification of the term part of the goal; but of the three subsequent subgoals that reflect the reassumption of the modified assumption terms, only the last one is correct: it shows the three new assumptions and the three implicit assumptions as desired. The other two subgoals reflect intermediate states of the computation in which certain assumptions are missing – neither implicit nor explicit, but held in temporary data structures.

Whether the current implementation of `SUBST_ALL_TAC` in HOL is the best one is not relevant here; nor is whether `SUBST_ALL_TAC` represents a 'good' style of reasoning. It is sufficient to note that, in this case, following the implementation is not a useful technique.

In any case, this account shown is flawed in two other ways: (i) the fact that the assumption `w < 5` is not affected by the substitution would be explained more clearly if that assumption were not said to have been processed like the others (although it is); and (ii) the account would be less tedious and if it did not report the processing of each assumption in sequence, but all together. The sequence results from the fact that although `POP_ASSUM_LIST` removes all of the assumptions at once, `ASSUME_TAC` th is not one of the HOL tactics for which a simultaneous version is provided (as `SUBST_TAC` is for `SUBST1_TAC`).

This suggests a second approach: namely, to implement a function called, say, `NAMED_ASSUME_LIST_TAC` that generalizes `NAMED_ASSUME_TAC`. `NAMED_ASSUME_LIST_TAC` computes the effect of adding a list of assumptions in sequence to a goal, then presents and justifies the result in one proof step, as though the assumptions had been added simultaneously. Implicit assumptions are recorded as a matter of course by the internal `ASSUME_TAC`'s. When the addition of the assumptions is packaged into one step with its own account, then `NAMED_SUBST_ALL` can then be implemented to satisfy

```
NAMED_SUBST_ALL_TAC rth =
NAMED_SUBST1_TAC rth THEN
NAMED_POP_ASSUM_LIST
  (\thl. NAMED_ASSUME_LIST_TAC [SUBS [rth] thn;
                                .
                                .
                                .
                                                ;
                                SUBS [rth] th1] )
```

so that its account spares the user the sequential computation of the re-assumptions. The account thus produced for the example is:

```
This is the proof of the conjecture
>> example5:
   "z = x"
   Assuming
    The fact1: "y = x"
    The fact2: "w > x"
    The fact3: "w < 5"

>>>> We substitute according to the following equality:
     |- x = 1.
     Thus, it is sufficient to prove:

>> "z = 1"
   Assuming
    The fact1: "y = x"
    The fact2: "w > x"
    The fact3: "w < 5"

>>>> We use the facts that
     y = x |- y = 1
     w > x |- w > 1
     w < 5 |- w < 5.
     It is sufficient to prove:

>> "z = 1"
   Assuming
    The added hypothesis: "y = 1"
    The added hypothesis: "w > 1"
    The added hypothesis: "w < 5"
   Assuming implicitly
    The hypothesis of the theorem used: "y = x"
    The hypothesis of the theorem used: "w > x"
    The hypothesis of the theorem used: "w < 5"

...
This establishes

y = 1, w > 1, w < 5 |- z = 1

This establishes

y = x, w > x, w < 5 |- z = 1

This establishes
```

```
y = x, w > x, w < 5 |- z = x

This completes the proof of the conjecture
>> example5:
   "z = x"
   Assuming
    The fact1: "y = x"
    The fact2: "w > x"
    The fact3: "w < 5"
```

This is a great improvement over the previous account in showing only two steps: the modification of the term, and the one-step modification of the assumptions. It also has the property that the theorems returned by the justifications respectively achieve the subgoals shown.

A minor flaw of this version is that there is no way, in passing control from `NAMED_POP_ASSUM_LIST` to `NAMED_ASSUME_LIST_TAC`, to make special provisions for particular assumptions which are not affected by substitution; thus `w < 5`, in the example, has to be treated in the same way as the others. This causes a slight obscurity in the account.

It was noted in Section 7.4.1 that `NAMED_POP_ASSUM` could be elaborated, in the case that `NAMED_POP_ASSUM` f had no net effect, to return the identity justification, and so omit the account of the re-assumption on the reduced goal. The corresponding generalization of `NAMED_POP_ASSUM_LIST` pertains when `NAMED_POP_ASSUM_LIST` f has no net effect – that is, when *all* the popped assumptions reappear intact and in order. This elaboration of `NAMED_POP_ASSUM_LIST` would only help with `NAMED_SUBST_ALL_TAC` where *no* assumption were affected by substitution; that is, the choice is between reporting the re-assumption of all the modified assumptions, or reporting nothing.

A more serious flaw is that in implementing `NAMED_SUBST_ALL_TAC` in a different manner than HOL's `SUBST_ALL_TAC`, it is not necessarily the case that the two computations are (in a suitable sense) equivalent – the account therefore might not be explaining the HOL proof. This would at least require an argument about the two computations.

The third (and last) approach we consider is to implement `NAMED_SUBST_ALL_TAC` itself as a unit function with a one-step acount. To compute its results, `NAMED_SUBST_ALL_TAC` analyzes the results of applying `SUBST_ALL_TAC` to the corresponding ordinary goal, and then presents the results as if derived in one stroke. As part of the presentation, unchanged assumptions are noticed

and presented as if no substitution had been attempted. The analysis stage allows the new assumptions as well as the implicit (old) assumptions to be named in a meaningful way (rather than in due course by `ASSUME_TAC`).

The implementation of `NAMED_SUBST_ALL_TAC` in terms of `SUBST_ALL_TAC` is not difficult, but it does involve a certain amount of internal analysis. The account produced is as follows:

```
This is the proof of the conjecture
>> example5:
   "z = x"
   Assuming
    The fact1: "y = x"
    The fact2: "w > x"
    The fact3: "w < 5"

>>>> We substitute according to the following equality:
     |- x = 1.
     (likewise restating any assumptions made thus far which involve "x").
     Thus, it is sufficient to prove:

>> "z = 1"
   Assuming
    The new fact1: "y = 1"
    The new fact2: "w > 1"
    The fact3: "w < 5"
   Assuming implicitly
    The old fact1: "y = x"
    The old fact2: "w > x"

...
This establishes

y = 1, w > 1, w < 5 |- z = 1

This establishes

y = x, w > x, w < 5 |- z = x

This completes the proof of the conjecture
>> example5:
   "z = x"
   Assuming
    The fact1: "y = x"
    The fact2: "w > x"
    The fact3: "w < 5"
```

This approach has the advantages of using the implementation of HOL's corresponding tactic in the usual way, so there is no issue of differing computations. It also allows for a clearer naming scheme in the account produced. Finally, it gives an opportunity to note the rather odd pattern of

reasoning being used, at the node in the subgoal-proof tree representing the `NAMED_SUBST_TAC` step (before the subgoal in the account).

However, there is a new difficulty: whereas, in the previous attempts at an account, the new assumptions assumptions (made by `NAMED_ASSUME_TAC`) automatically caused the implicit assumptions to be recorded, there is no way to do this given only the results of the ordinary `SUBST_ALL_TAC`. Instead, the implicit assumptions (i.e. those original assumptions which would be affected by substitution) have to be identified and added as part of the presentation. Thus, there is again an argument to be made that HOL's behaviour is reflected here: it has to be argued that the direct implementation of `NAMED_SUBST_ALL_TAC` produces the same implicit assumptions that can be observed by experiment in HOL itself.

Whether the second or the third approach is best is difficult to say, but in any case, the first approach is clearly not adequate.

`NAMED_SUBST_ALL_TAC` *th* is the first example of a named tactic with a complex implementation (p. ...). A meaningful account neither parallels the HOL implementation of the ordinary tactic nor follows directly from it, but requires some new function to be implemented directly (`NAMED_ASSUME_LIST_TAC`, in the second approach, or `NAMED_SUBST_ALL_TAC` itself, in the third). The next such example are the strip functions (Section ...).

8 Continuations

The HOL functions in the next group to be considered also produce new tactics from old, as do the functions in the previous chapter. The members of this group differ from functions such as `POP_ASSUM` in that they all cause some inference to be done behind the scenes, and they can also affect the term parts of goals, in addition to the assumptions. The concealed inferences give the effect of performing two proof steps in one. The difficulty in giving accounts for these functions is to explain the concealed inferences coherently.

8.1 The Disjunctive Transformer

A typical example is `DISJ_CASES_THEN`, which maps a function f of type `thm -> tactic` and a disjunctive theorem to a new tactic. For the sake of

example, suppose that a new type, `:voltage`, has been introduced, with exactly two values, `hi` and `lo`. The new type is characterized by:

```
|- !(v:voltage). (v = hi) \/ (v = lo)
```

Suppose also that there is an operator, `AND`, such that

```
|- hi AND hi = hi
```

and

```
|- lo AND lo = lo
```

The effect of `DISJ_CASES_THEN` is illustrated below The goal is to show (for all `v`) that `v AND v = v`, given that `hi AND hi = hi` and `lo AND lo = lo`.

```
#let g = [],"v AND v = v";;
g = ([], "v AND v = v") : (* list # term)
...
th = |- (v = hi) \/ (v = lo)
#let gl,p = DISJ_CASES_THEN SUBST1_TAC th g;;
gl = [([], "hi AND hi = hi"); ([], "lo AND lo = lo")] : goal list
p = - : proof
...
#th1 =          |- hi AND hi = hi
 th2 =          |- lo AND lo = lo
 th1' = v = hi |- hi AND hi = hi
 th2' = v = lo |- lo AND lo = lo
#p[th1;th2];;
|- v AND v = v
#p[th1';th2'];;
|- v AND v = v
```

In the example, the new tactic `DISJ_CASES_THEN SUBST1_TAC th` maps the goal to two subgoals by extracting from the disjunctive theorem

```
|- (v = hi) \/ (v = lo)
```

the two disjunct terms, `v = hi` and `v = lo`, assuming these, and using the two resulting theorems – in parallel – as parameters to two applications of the substitution function. The two new subgoals are the values of

```
SUBST1_TAC (ASSUME "v = hi") g
```

and

```
SUBST1_TAC (ASSUME "v = lo") g
```

The subgoals carry the implicit assumptions `v = hi` and `v = lo` respectively; these are introduced, in each case, by the act of assuming the disjunct term. The justification (`p`) relies on (i) the inference rule for substitution (see Section ...) and (ii) the rule for disjunction (`DISJ_CASES`, see Description ...). The substitution rule adds the respective assumptions to the two achieving theorems if they are not already present:

```
...
th1 = |- hi AND hi = hi
th2 = |- lo AND lo = lo

#let gl1,p1 = SUBST1_TAC (ASSUME "v = hi") g;;
gl1 = [([], "hi and hi = hi")] : goal list
p1 = - : proof

#print_all_thm(p1[th1]);;
v = hi |- v and v = v

#let gl2,p2 = SUBST1_TAC (ASSUME "v = lo") g;;
gl2 = [([], "lo and lo = lo")] : goal list
p2 = - : proof

#print_all_thm(p2[th2]);;
v = lo |- v and v = v
```

The disjunction rule then dismisses the two added assumptions as it combines the two achieving theorems to yield the theorem achieving `g`:

```
#print_all_thm(DISJ_CASES th (p1[th1]) (p2[th2]));;
|- v AND v = v
```

The addition of the implicit assumptions to the subgoals does not depend on the function f to which `DISJ_CASES_THEN` is applied, but rather, on the assumptions being made at all; for example, using the function `K ALL_TAC` to throw away the assumed terms, as in Section ..., we have the following results (having established above, for all `v`, that `|- v AND v = v`):

```
#let gl,p = DISJ_CASES_THEN (K ALL_TAC) th g;;
gl = [([], "v AND v = v"); ([], "v AND v = v")] : goal list
p = - : proof

...

#th1'' =          |- v AND v = v
 th2'' =          |- v AND v = v
 th1''' = v = hi |- v AND v = v
 th2''' = v = lo |- v AND v = v

#p[th1'';th2''];;
|- v AND v = v

#p[th1''';th2'''];;
|- v AND v = v
```

In any case, the tactic `DISJ_CASES_THEN SUBST1_TAC th`, in one step, splits a goal into two subgoals by applying two distinct substitutions – based on the disjuctive theorem `th` – in parallel to the original goal. In this one-step process, the assumptions `v = hi` and `v = lo` do not appear explicitly; they are added and then dismissed only behind the scenes, when the justification function is applied. This one-step process shown below is more elegant than the straightforward two-step process shown below, as the latter (i) requires explicit reference to the terms `v = hi` and `v = lo`, and (ii) leaves the two 'used' assumptions in the respective subgoals after the substitutions based on them have been made:

```
#let gl3,p3 = DISJ_CASES_TAC th g;;
gl3 =
[(["v = hi"], "v AND v = v"); (["v = lo"], "v AND v = v")]
: goal list
p3 = - : proof

#let gl4,p4 = SUBST1_TAC(ASSUME "v = hi")(hd gl3);;
gl4 = [(["v = hi"], "hi AND hi = hi")] : goal list
p4 = - : proof

#let gl5,p5 = SUBST1_TAC(ASSUME "v = lo")(hd(tl gl3));;
gl5 = [(["v = lo"], "lo AND lo = lo")] : goal list
p5 = - : proof

#let th4 = p4[th1];;
th4 = . |- v AND v = v

#print_all_thm th4;;
v = hi |- v AND v = v

#let th5 = p5[th2];;
th5 = . |- v AND v = v
```

```
#print_all_thm th5;;
v = lo |- v AND v = v
#print_all_thm(p3[th4;th5]);;
|- v AND v = v
```

From the viewpoint of accounts, however, the one-step tactic presents difficulties. It was possible (Section ...) to report the tactic `POP_ASSUM SUBST1_TAC` in one step, as a substitution. That was possible because the tactic transformer `POP_ASSUM` simply supplied the argument for an application of `NAMED_SUBST1_TAC` to an amended goal. The tactic `DISJ_CASES_THEN SUBST1_TAC th`, in contrast, cannot be explained clearly in one step (e.g. as a substitution), because it consists internally of a disjunctive split into two identical subgoals *followed by* distinct and parallel substitutions on two 'copies' of the original goal.

In the current example, what has to be explained is the move from the named goal (`ng`, say)

```
>> "v AND v = v"
```

to the two named subgoals

```
>> "hi AND hi = hi"
```

and

```
>> "lo AND lo = lo"
```

and this move is not explained by any single existing tactic.

Even by devoting a node in the subgoal-proof tree to the application of compound tactics of the form `NAMED_DISJ_CASES_THEN` f th, so that there is an opportunity for choosing a wording to explain the disjunctive split, a coherent account still cannot be produced. (This is demonstrated below.)

To devote a node in this way, `NAMED_DISJ_CASES_THEN` is implemented in parallel with the HOL implementation of the ordinary tactic `DISJ_CASES_THEN` f th. The proof step of the node is identified by a string, say `'NAMED_DISJ_CASES_THEN'`. In the example case,

```
NAMED_DISJ_CASES_THEN NAMED_SUBST1_TAC th ng
```

would compute

```
NAMED_SUBST1_TAC (ASSUME "v = hi") ng
```

and

```
NAMED_SUBST1_TAC (ASSUME "v = lo") ng
```

and then use the pair of resulting subgoals and justifications to construct the justification. The justification is the function which when given the two respective sub-accounts returns an account consisting of (i) the single combined proof step, (ii) the two subgoals, (iii) the two sub-accounts, and (iv) the method for computing the achieving theorem: namely, by applying the two justifications respectively to the two sub-accounts, selecting the two theorems from within these accounts, and combining these theorems to justify the disjunctive split. The account thus produced is:

```
This is the proof of the conjecture
>> example1:
   "v AND v = v"

>>>> We consider the two cases suggested by the fact
     |- (v = hi) \/ (v = lo),
     namely
     v = hi |- v = hi
     and
     v = lo |- v = lo
     It is thus sufficient to prove the following:

>> left disjunctive case:
   "hi AND hi = hi"
   Assuming implicitly
    The hypothesis of the equality: "v = hi"

>> right disjunctive case:
   "lo AND lo = lo"
   Assuming implicitly
    The hypothesis of the equality: "v = lo"

The proof of the
>> left disjunctive case:
   "hi AND hi = hi"
   Assuming implicitly
    The hypothesis of the equality: "v = hi"

is as follows:

...

This establishes

|- hi AND hi = hi

The proof of the
>> right disjunctive case:
```

```
   "lo AND lo = lo"
   Assuming implicitly
    The hypothesis of the equality: "v = lo"
is as follows:
...
This establishes
|- lo AND lo = lo
This establishes
|- v AND v = v
This completes the proof of the conjecture
>> example1:
   "v AND v = v"
```

The problem with this account is that although it explains the disjunctive split, it does not provide any opportunity for reporting or explaining the substitutions; the node that is constructed for the compound step branches directly into the two subgoals, each with an account of its own. The substitutions are justified, internally to the tactic, as part of the combined justification. The only evidence in the account that any substitutions took place is the move from the term `v AND v = v` to the terms `hi AND hi = hi` and `lo AND lo = lo` – and the implicit assumption that is introduced in each case. Accounts of the substitutions are thus not part of the account of the combined step.

In this case, it might be possible for a user to guess that the unexplained step was substitution, but it might not be possible to guess for a more complex function than substitution.

The account produced in this way becomes even more obscure when one of the subgoals is actually solved by the concealed step. In the schematic example below, the function `\th.NAMED_REWRITE_TAC[th]` is used in place of `NAMED_SUBST1_TAC` so that one of the subgoals can be solved. (`P` is some property true of `lo`.)

```
This is the proof of the conjecture
>> example2:
   "(v = hi) \/ P v"
>>>> We consider the two cases suggested by the fact
     |- (v = hi) \/ (v = lo),
     namely
     v = hi |- v = hi
     and
```

```
     v = lo |- v = lo
     It is thus sufficient to prove the following:

>> "(lo = hi) \/ P lo"
   Assuming implicitly
    The hypothesis of the equality: "v = lo"

...

This establishes

|- (lo = hi) \/ P lo

This establishes

|- (v = hi) \/ P v

This completes the proof of the conjecture
>> example2:
   "(v = hi) \/ P v"
```

In this account, the `v = hi` subgoal is solved internally (by rewriting) without ever having been displayed; so as well as the unexplained function (rewriting), the missing case and the the way in which the function solved the missing case would also have to be guessed. The point also applies where *both* cases are generated and solved internally by the combined tactic. A trivial example illustrates this:

```
This is the proof of the conjecture
>> example3:
   "(v = hi) \/ (v = lo)"

>>>> This follows by considering the two cases suggested by the fact
     |- (v = hi) \/ (v = lo),
     namely
     v = hi |- v = hi
     and
     v = lo |- v = lo

This establishes

|- (v = hi) \/ (v = lo)

This completes the proof of the conjecture
>> example3:
   "(v = hi) \/ (v = lo)"
```

To give a clear account of a tactic of the form `NAMED_DISJ_CASES_THEN` *f th*, it is therefore necessary to generate more than one node of the subgoal-proof tree. The disjunctive split is accorded a node of its own, and this branches into a node for each application of the second tactic. Thus an account attaches to the disjunction node, as well as to each of the daughter nodes; so all steps are explained.

In the framework of proof accounts, a node represents the application of a tactic to a goal to produce subgoals and a justification. Without altering the basic framework, this means that the disjunctive split has to be regarded as the application of a tactic. One possibility is to use the existing named tactic `NAMED_DISJ_CASES_TAC` *th* to implement the split.

The effect of applying the straightforward disjunction tactic is simply to create two subgoals with the respective disjuncts as explicit assumptions. To produce the same end result as the tactic `NAMED_DISJ_CASES_THEN` *f th*, the tactic `NAMED_DISJ_CASES_TAC` *th* must be sequenced with a tactic which in each case removes the new explicit assumption term from each subgoal, assumes it, passes the resulting theorem as paramaters to *f*, and applies the resulting tactic to the subgoal.

This suggests a popping operation. Furthermore, it suggests a popping operation which necessarily keeps the popped term as an implicit assumption, since, by its implementation, an application of the tactic `DISJ_CASES_THEN` *f th* to a goal always adds the respective disjunct terms of the conclusion of *th* as implicit assumptions to its two resulting subgoals. (Insisting on keeping the popped term only makes a difference where *f* has the property of throwing away its theorem parameter, e.g. where *f* is `K NAMED_ALL_TAC`. For the purpose of succinct printing of accounts in this chapter, we will not insist on keeping the popped term – the issue of lost assumptions does not arise in any of the examples.)

If we define `NAMED_DISJ_CASES_THEN` *f th* to be `NAMED_DISJ_CASES_TAC` *th* `THEN NAMED_POP_TRACE` *f* (see Section ...), then the account produced in the example case is as shown below. (Since `NAMED_DISJ_CASES_TAC` *th* produces two subgoals, the sequencer `THEN` causes `NAMED_POP_TRACE` *f* to be applied to each.)

```
This is the proof of the conjecture
>> example1:
   "v AND v = v"

>>>> We consider the two cases suggested by the fact
     |- (v = hi) \/ (v = lo)

>> left disjunct case:
   "v AND v = v"
   Assuming
    The left disjunct: "v = hi"

>> right disjunct case:
   "v AND v = v"
```

```
   Assuming
    The right disjunct: "v = lo"

The proof of the
>> left disjunct case:
   "v AND v = v"
   Assuming
    The left disjunct: "v = hi"

is as follows:
>>>> We substitute according to the following equality:
     v = hi |- v = hi.
     Thus, it is sufficient to prove:

>> "hi AND hi = hi"
   Assuming implicitly
    The hypothesis of the equality: "v = hi"
    The left disjunct: "v = hi"

...

This establishes

|- hi AND hi = hi

This establishes

v = hi |- v AND v = v

The proof of the
>> right disjunct case:
   "v AND v = v"
   Assuming
    The right disjunct: "v = lo"

is as follows:
>>>> We substitute according to the following equality:
     v = lo |- v = lo.
     Thus, it is sufficient to prove:

>> "lo AND lo = lo"
   Assuming implicitly
    The hypothesis of the equality: "v = lo"
    The right disjunct: "v = lo"

...

This establishes

|- lo AND lo = lo

This establishes

v = lo |- v AND v = v

This establishes

|- v AND v = v

This completes the proof of the conjecture
>> example1:
   "v AND v = v"
```

This seems a reasonable account.

When the function f is `NAMED_ASSUME_TAC`, the mechanism internal to the named popping functions, described in Section ..., automatically assures that there is no unnecessary accounting; the account of `NAMED_DISJ_CASES_THEN NAMED_ASSUME_TAC th` on `ng` is:

```
This is the proof of the conjecture
>> example1:
   "v AND v = v"

>>>> We consider the two cases suggested by the fact
     |- (v = hi) \/ (v = lo)

>> left disjunct case:
   "v AND v = v"
   Assuming
    The left disjunct: "v = hi"

>> right disjunct case:
   "v AND v = v"
   Assuming
    The right disjunct: "v = lo"

The proof of the
>> left disjunct case:
   "v AND v = v"
   Assuming
    The left disjunct: "v = hi"

is as follows:

...

This establishes

v = hi |- v AND v = v

The proof of the
>> right disjunct case:
   "v AND v = v"
   Assuming
    The right disjunct: "v = lo"

is as follows:

...

This establishes

v = lo |- v AND v = v

This establishes

|- v AND v = v

This completes the proof of the conjecture
>> example1:
   "v AND v = v"
```

8.2 Implementation Issues

The only real fault of the scheme described above is its inefficiency. This results from the fact that, in HOL, transformers such as `DISJ_CASES_THEN` are taken as primitives, and tactics such as `DISJ_CASES_TAC` *th* are elaborations of the primitives. Thus, `DISJ_CASES_TAC` is implemented as `DISJ_CASES_THEN` applied to `ASSUME_TAC`. The implementation of the named functions, as described in Section 8.1, reverses HOL's order of dependency. Thus, unfortunately, the computation of `NAMED_DISJ_CASES_THEN` requires `NAMED_DISJ_CASES_TAC` to be computed, which requires `DISJ_CASES_TAC`, which requires `DISJ_CASES_THEN`; two translations are made, internally, to produce the desired account.

HOL's particular choice of primitive functions is useful for implementation purposes, and it also provides the user of the system with tactic-building tools rather than with specific tactics; variations of `DISJ_CASES_TAC` are defined easily via `DISJ_CASES_THEN`. However, the HOL system is not generally presented or learned in the implementation's order of dependency; simple tactics usually are presented first and 'advanced' functions later. Thus, for many users, it probably seems natural to regard `DISJ_CASES_TAC` as the primitive function and `DISJ_CASES_THEN` as the elaboration, as is done for producing accounts.

In any case, functions such as `NAMED_DISJ_CASES_TAC` could be implemented directly, rather than in terms of `DISJ_CASES_TAC` (and hence of `DISJ_CASES_THEN` and `ASSUME_TAC`). This option involves more work to implement, but the main objection to it is that it makes it less clear that the same proof is being performed as in the ordinary system. Confidence would require an argument that the same inference chains were generated either way.

8.3 Other Transformers which Introduce Assumptions

The method for implementing `NAMED_DISJ_CASES_THEN` can be applied to several other tactic transformers in HOL which similarly cause implicit assumptions to be generated.

8.3.1 The Discharging Transformer

By implemenating `NAMED_DISCH_THEN` f as `NAMED_DISCH_TAC THEN NAMED_POP_TRACE` f, a comprehensible two-step account is produced for a one-step tactic.

The effect of the transformer `DISCH_THEN` is illustrated below. For example, for the goal

```
g = ([], "(v = hi) ==> (v AND v = v)")
```

we have, in one step,

```
#let gl,p = DISCH_THEN SUBST1_TAC g;;
gl = [([], "hi AND hi = hi")] : goal list
p = - : proof
```

where:

```
...
th1 =   |- hi AND hi = hi
th2 = v = hi |- hi AND hi = hi
#p[th1];;
|- (v = hi) ==> (v AND v = v)
#p[th2];;
|- (v = hi) ==> (v AND v = v)
```

Under the implementation suggested, the two-step account of the one-step tactic (which introduces an implicit assumption) is as follows:

```
This is the proof of the conjecture
>> example4:
   "(v = hi) ==> (v AND v = v)"
>>>> It is sufficient to prove:
>> "v AND v = v"
   Assuming
    The antecedent: "v = hi"
>>>> We substitute according to the following equality:
     v = hi |- v = hi.
     Thus, it is sufficient to prove:
>> "hi AND hi = hi"
   Assuming implicitly
    The hypothesis of the equality: "v = hi"
    The antecedent: "v = hi"
...
This establishes
|- hi AND hi = hi
This establishes
```

```
v = hi |- v AND v = v
```

This establishes

```
|- (v = hi) ==> (v AND v = v)
```

This completes the proof of the conjecture

```
>> example4:
   "(v = hi) ==> (v AND v = v)"
```

8.3.2 The Choice Transformer

Analogously, by implemenating `NAMED_CHOOSE_THEN` f as `NAMED_CHOOSE_TAC THEN NAMED_POP_TRA` f, a comprehensible two-step account is produced for a one-step tactic.

The following schematic example illustrates the use of `CHOOSE_THEN`, using the fact that (for all `y`) `|- ?x. y = PRE x`). (`Q` is some property true of all numbers.)

```
...
th = |- ?x. y = PRE x
#let g = [], "(Q:num -> bool) y";;
g = ([], "Q y") : (* list # term)
#let gl,p = CHOOSE_THEN SUBST1_TAC th g;;
gl = [([], "Q(PRE x)")] : goal list
p = - : proof
...
thm =             |- Q(PRE x)
thm' = y = PRE x |- Q(PRE x)
#p[thm];;
|- Q y
#p[thm'];;
|- Q y
```

Like `DISJ_CASES_THEN`, `CHOOSE_THEN` f introduces an implicit assumption; in this case, `y = PRE x`, the assumption about the witness constant.

The implementation of `NAMED_CHOOSE_THEN` f as `NAMED_CHOOSE_TAC THEN NAMED_POP_TRACE` f gives the following two-step account for the example:

```
This is the proof of the conjecture
>> example5:
   "Q y"
>>>> Using the term "x"
     as a witness to the fact
```

```
     |- ?x. y = PRE x
     it is sufficient to prove:

>> "Q y"
   Assuming
    The witness hypothesis: "y = PRE x"

>>>> We substitute according to the following equality:
     y = PRE x |- y = PRE x.
     Thus, it is sufficient to prove:

>> "Q(PRE x)"
   Assuming implicitly
    The hypothesis of the equality: "y = PRE x"
    The witness hypothesis: "y = PRE x"

...

This establishes

|- Q(PRE x)

This establishes

y = PRE x |- Q y

This establishes

|- Q y

This completes the proof of the conjecture
>> example5:
   "Q y"
```

This again seems a reasonable explanation.

8.4 Transformers which do not Introduce Assumptions

The transformers that do not introduce implicit assumptions are `CONJUNCTS_THEN` and the resolution functions `IMP_RES_THEN` and `RES_THEN`. A different approach is used for these than for the others.

8.4.1 The Conjunction Transformer

The transformer `CONJUNCTS_THEN` is different from those described thus far in that it does not introduce implicit assumptions. Given a conjunctive theorem, it is possible to *infer* the two conjuncts immediately. Hence, neither of the two conjunct terms (nor the conjunctive term itself) has to be assumed implicitly during the decomposition of the goal (and hence dismissed later when the justification of the conjunctive split is applied). The inference *could*

be deferred in this way, but there is a small economy of inference steps in not doing so.

The effect of CONJUNCTS_THEN is illustrated by the following example, using a consequence of the fact |- (!n. 0 + n = n) /\ (!m n. (SUC m) + n = SUC(m + n)):

```
...
th = |- (0 + n = n) /\ ((SUC m) + n = SUC(m + n))

#let g = [],"(SUC m) + n = SUC(m + (0 + n))";;
g = ([], "(SUC m) + n = SUC(m + (0 + n))") : (* list # term)

#let gl,p = CONJUNCTS_THEN SUBST1_TAC th g;;
gl = [([], "SUC(m + n) = SUC(m + n)")] : goal list
p = - : proof

...
thm =                                                |- SUC(m + n) = SUC(m + n)
thm' = 0 + m = m, (SUC m) + n = SUC(m + n)           |- SUC(m + n) = SUC(m + n)
thm'' = (0 + n = n) /\ ((SUC m) + n = SUC(m + n)) |- SUC(m + n) = SUC(m + n)

#p[thm];;
|- (SUC m) + n = SUC(m + (0 + n))

#print_all_thm(p[thm']);;
0 + m = m, (SUC m) + n = SUC(m + n) |- (SUC m) + n = SUC(m + (0 + n))

#print_all_thm(p[thm'']);;
(0 + n = n) /\ ((SUC m) + n = SUC(m + n))
|- (SUC m) + n = SUC(m + (0 + n))
```

As illustrated, neither of the the conjuncts nor the conjunction is an implicit assumption of the subgoal.

As it happens, there is no function 'CONJUNCTS_TAC', analogous to DISJ_CASES_TAC, provided in HOL. CONJUNCTS_TAC th, by analogy, would be defined as CONJUNCTS_THEN ASSUME_TA in the above example, this would return, in one step, the subgoal

```
["(SUC m) + n = SUC(m + n)"; "0 + m = m"],
 "(SUC(0 + m)) + n = SUC(m + n)"
```

It might seem useful to define the function NAMED_CONJUNCTS_TAC so that NAMED_CONJUNCTS_THEN could be defined in terms of it, by analogy with NAMED_DISJ_CASES_THEN and the others. However, no fuction NAMED_CONJUNCTS_TAC that introduces assumptions can support a NAMED_CONJUNCTS_THEN that satisfactorily models CONJUNCTS_THEN, since CONJUNCTS_THEN does not introduce any (explicit or implicit) assumptions.

To illustrate this point, it is easy to implement a `NAMED_CONJUNCTS_TAC` which adds the conjuncts (and justifies the additions). The account of that much, in the example case, is:

```
This is the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"

>>>> We use the two separate theorems implied by the fact
     |- (0 + n = n) /\ ((SUC m) + n = SUC(m + n)).
     It is thus sufficient to prove:

>> "(SUC m) + n = SUC(m + (0 + n))"
   Assuming
    The second conjunct: "(SUC m) + n = SUC(m + n)"
    The first conjunct: "0 + n = n"

...

This establishes

(SUC m) + n = SUC(m + n), 0 + n = n |- (SUC m) + n = SUC(m + (0 + n))

This establishes

|- (SUC m) + n = SUC(m + (0 + n))

This completes the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"
```

If the function `NAMED_CONJUNCTS_THEN` were now defined as `NAMED_CONJUNCTS_TAC` followed by two popping operations in sequence, the account of

```
NAMED_CONJUNCTS_THEN NAMED_SUBST1_TAC th ng
```

in the example case, is:

```
This is the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"

>>>> We use the two separate theorems implied by the fact
     |- (0 + n = n) /\ ((SUC m) + n = SUC(m + n)).
     It is thus sufficient to prove:

>> "(SUC m) + n = SUC(m + (0 + n))"
   Assuming
    The second conjunct: "(SUC m) + n = SUC(m + n)"
    The first conjunct: "0 + n = n"

>>>> We substitute according to the following equality:
     (SUC m) + n = SUC(m + n) |- (SUC m) + n = SUC(m + n).
     Thus, it is sufficient to prove:
```

```
>> "SUC(m + n) = SUC(m + (0 + n))"
   Assuming
    The first conjunct: "0 + n = n"
   Assuming implicitly
    The hypothesis of the equality: "(SUC m) + n = SUC(m + n)"
    The second conjunct: "(SUC m) + n = SUC(m + n)"

>>>> We substitute according to the following equality:
     0 + n = n |- 0 + n = n.
     Thus, it is sufficient to prove:

>> "SUC(m + n) = SUC(m + n)"
   Assuming implicitly
    The hypothesis of the equality: "0 + n = n"
    The first conjunct: "0 + n = n"
    The hypothesis of the equality: "(SUC m) + n = SUC(m + n)"
    The second conjunct: "(SUC m) + n = SUC(m + n)"

...

This establishes

|- SUC(m + n) = SUC(m + n)

This establishes

0 + n = n |- SUC(m + n) = SUC(m + (0 + n))

This establishes

0 + n = n, (SUC m) + n = SUC(m + n) |- (SUC m) + n = SUC(m + (0 + n))

This establishes

|- (SUC m) + n = SUC(m + (0 + n))

This completes the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"
```

This is a good account in that it is in three steps: the conjunctive split and the two sequential substitutions. The accounts of the substitutions are produced directly via the function `NAMED_SUBST1_TAC`. The inference chain generated is arguably the same as that generated by `CONJUNCTS_THEN SUBST1_TAC th g`, with the addition of the inferences in which the added assumptions are introduced and dismissed. However, the subgoal thus carries `0 + n = n` and `(SUC m) + n = SUC(m + n)` as implicit assumptions, which is not satisfactory.

In the account of

```
CONJUNCTS_THEN NAMED_ASSUME_TAC th ng
```

implicit asusmptions are not an issue; and the account produced in the same

way as the above is therefore satisfactory. It is also concise because, internally, the popping function notices and omits the pop and re-assume steps:

```
This is the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"

>>>> We use the two separate theorems implied by the fact
     |- (0 + n = n) /\ ((SUC m) + n = SUC(m + n)).
     It is thus sufficient to prove:

>> "(SUC m) + n = SUC(m + (0 + n))"
   Assuming
    The second conjunct: "(SUC m) + n = SUC(m + n)"
    The first conjunct: "0 + n = n"

...

This establishes

(SUC m) + n = SUC(m + n), 0 + n = n |- (SUC m) + n = SUC(m + (0 + n))

This establishes

|- (SUC m) + n = SUC(m + (0 + n))

This completes the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"
```

However, a more serious defect of this implementation of `NAMED_CONJUNCTS_THEN` is that the sequential popping operations produce the wrong effect in contexts in which the assumption stack is disturbed by the first popping operation (which may itself involve further transformers) before the second takes place. (This sort of disturbance is a general problem in the stack approach, and was a factor motivating the development

of the transformer functions.)

The defect can be repaired by taking `NAMED_CONJUNCTS_TAC` simply to be `NAMED_ASSUME_TAC`, and `NAMED_CONJUNCTS_THEN` f th to `NAMED_CONJUNCTS_TAC` th followed by the popping of the whole added conjunction – to a function that infers the two separate theorems, and then applies f to the two theorems in sequence. The account of the example, under this interpretation, is:

```
This is the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"

>>>> We use the fact that
     |- (0 + n = n) /\ ((SUC m) + n = SUC(m + n)).
     It is sufficient to prove:
```

```
>> "(SUC m) + n = SUC(m + (0 + n))"
   Assuming
    The added hypothesis: "(0 + n = n) /\ ((SUC m) + n = SUC(m + n))"

>>>> We substitute according to the following equality:
     (0 + n = n) /\ ((SUC m) + n = SUC(m + n)) |- 0 + n = n.
     Thus, it is sufficient to prove:

>> "(SUC m) + n = SUC(m + n)"
   Assuming implicitly
    The hypothesis of the equality: "(0 + n = n) /\
                                     ((SUC m) + n = SUC(m + n))"
    The added hypothesis: "(0 + n = n) /\ ((SUC m) + n = SUC(m + n))"

>>>> We substitute according to the following equality:
     (0 + n = n) /\ ((SUC m) + n = SUC(m + n))
     |- (SUC m) + n = SUC(m + n).
     Thus, it is sufficient to prove:

>> "SUC(m + n) = SUC(m + n)"
   Assuming implicitly
    The hypothesis of the equality: "(0 + n = n) /\
                                     ((SUC m) + n = SUC(m + n))"
    The added hypothesis: "(0 + n = n) /\ ((SUC m) + n = SUC(m + n))"

...

This establishes

|- SUC(m + n) = SUC(m + n)

This establishes

(0 + n = n) /\ ((SUC m) + n = SUC(m + n)) |- (SUC m) + n = SUC(m + n)

This establishes

(0 + n = n) /\ ((SUC m) + n = SUC(m + n))
|- (SUC m) + n = SUC(m + (0 + n))

This establishes

|- (SUC m) + n = SUC(m + (0 + n))

This completes the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"
```

Here, the conjunction is mentioned, if not split, in one step, and the substitutions have adequate accounts of their own. This avoids the defect of the previous method, but it still, likewise, generates a undesired implicit assumption. In addition, the account of `NAMED_CONJUNCTS_THEN NAMED_ASSUME_TAC` is now more awkward, since there is no pop and re-assume step to omit:

```
This is the proof of the conjecture
>> example7:
```

```
   "(SUC m) + n = SUC(m + (0 + n))"

>>>> We use the fact that
     |- (0 + n = n) /\ ((SUC m) + n = SUC(m + n)).
     It is sufficient to prove:

>> "(SUC m) + n = SUC(m + (0 + n))"
   Assuming
    The added hypothesis: "(0 + n = n) /\ ((SUC m) + n = SUC(m + n))"

>>>> We use the fact that
     (0 + n = n) /\ ((SUC m) + n = SUC(m + n)) |- 0 + n = n.
     It is sufficient to prove:

>> "(SUC m) + n = SUC(m + (0 + n))"
   Assuming
    The added hypothesis: "0 + n = n"
   Assuming implicitly
    The hypothesis of the theorem used: "(0 + n = n) /\
                                         ((SUC m) + n = SUC(m + n))"
    The added hypothesis: "(0 + n = n) /\ ((SUC m) + n = SUC(m + n))"

>>>> We use the fact that
     (0 + n = n) /\ ((SUC m) + n = SUC(m + n))
     |- (SUC m) + n = SUC(m + n).
     It is sufficient to prove:

>> "(SUC m) + n = SUC(m + (0 + n))"
   Assuming
    The added hypothesis: "(SUC m) + n = SUC(m + n)"
    The added hypothesis: "0 + n = n"
   Assuming implicitly
    The hypothesis of the theorem used: "(0 + n = n) /\
                                         ((SUC m) + n = SUC(m + n))"
    The added hypothesis: "(0 + n = n) /\ ((SUC m) + n = SUC(m + n))"

...

This establishes

(SUC m) + n = SUC(m + n), 0 + n = n |- (SUC m) + n = SUC(m + (0 + n))

This establishes

0 + n = n, (0 + n = n) /\ ((SUC m) + n = SUC(m + n))
|- (SUC m) + n = SUC(m + (0 + n))

This establishes

(0 + n = n) /\ ((SUC m) + n = SUC(m + n))
|- (SUC m) + n = SUC(m + (0 + n))

This establishes

|- (SUC m) + n = SUC(m + (0 + n))

This completes the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"
```

In both interpretations discussed so far, undesired implicit assumptions are added to the subgoal. Omitting the `NAMED_CONJUNCTS_TAC` step, which causes this problem, is still not a good solution; this time, because it obscures the origin of the conjuncts:

```
This is the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"

>>>> We substitute according to the following equality:
     |- 0 + n = n.
     Thus, it is sufficient to prove:

>> "(SUC m) + n = SUC(m + n)"

>>>> We substitute according to the following equality:
     |- (SUC m) + n = SUC(m + n).
     Thus, it is sufficient to prove:

>> "SUC(m + n) = SUC(m + n)"

...

This establishes

|- SUC(m + n) = SUC(m + n)

This establishes

|- (SUC m) + n = SUC(m + n)

This establishes

|- (SUC m) + n = SUC(m + (0 + n))

This completes the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"
```

The only remaining solution would seem to be to include a step in which the conjunction is at least mentioned, but in which no assumptions are added. In the current framework, this requires that the account of the first step include a subgoal, albeit unchanged from the previous subgoal. The account by this method is not therefore perfectly tidy, but does at least model HOL's `CONJUNCTS_THEN` function:

```
This is the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"

>>>> We use the two separate theorems implied by the fact
     |- (0 + n = n) /\ ((SUC m) + n = SUC(m + n)).
     The two theorems are used in sequence.  We are showing:
```

```
>> "(SUC m) + n = SUC(m + (0 + n))"

>>>> We substitute according to the following equality:
     |- 0 + n = n.
     Thus, it is sufficient to prove:

>> "(SUC m) + n = SUC(m + n)"

>>>> We substitute according to the following equality:
     |- (SUC m) + n = SUC(m + n).
     Thus, it is sufficient to prove:

>> "SUC(m + n) = SUC(m + n)"

...

This establishes

|- SUC(m + n) = SUC(m + n)

This establishes

|- (SUC m) + n = SUC(m + n)

This establishes

|- (SUC m) + n = SUC(m + (0 + n))

This establishes

|- (SUC m) + n = SUC(m + (0 + n))

This completes the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"
```

In the event of f being `NAMED_ASSUME_TAC`, the account is now:

```
This is the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"

>>>> We use the two separate theorems implied by the fact
     |- (0 + n = n) /\ ((SUC m) + n = SUC(m + n)).
     The two theorems are used in sequence.  We are showing:

>> "(SUC m) + n = SUC(m + (0 + n))"

>>>> We use the fact that
     |- 0 + n = n.
     It is sufficient to prove:

>> "(SUC m) + n = SUC(m + (0 + n))"
   Assuming
    The added hypothesis: "0 + n = n"

>>>> We use the fact that
     |- (SUC m) + n = SUC(m + n).
     It is sufficient to prove:

>> "(SUC m) + n = SUC(m + (0 + n))"
```

```
   Assuming
    The added hypothesis: "(SUC m) + n = SUC(m + n)"
    The added hypothesis: "0 + n = n"
...
This establishes
(SUC m) + n = SUC(m + n), 0 + n = n |- (SUC m) + n = SUC(m + (0 + n))
This establishes
0 + n = n |- (SUC m) + n = SUC(m + (0 + n))
This establishes
|- (SUC m) + n = SUC(m + (0 + n))
This establishes
|- (SUC m) + n = SUC(m + (0 + n))
This completes the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"
```

A minor refinement of this solution is to implement `NAMED_CONJUNCTS_THEN` to notice when f is effectively the same as `NAMED_ASSUME_TAC`, and where it is, to use instead a trivial variant of `NAMED_ASSUME_TAC` which labels the new assumptions as conjuncts. (The point of this refinement is made clear in Section ...). The previous account is now:

```
This is the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"
>>>> We use the two separate theorems implied by the fact
     |- (0 + n = n) /\ ((SUC m) + n = SUC(m + n)).
     The two theorems are used in sequence.  We are showing:
>> "(SUC m) + n = SUC(m + (0 + n))"
>>>> We use the fact that
     |- 0 + n = n.
     It is sufficient to prove:
>> "(SUC m) + n = SUC(m + (0 + n))"
   Assuming
    The left conjunct: "0 + n = n"
>>>> We use the fact that
     |- (SUC m) + n = SUC(m + n).
     It is sufficient to prove:
>> "(SUC m) + n = SUC(m + (0 + n))"
   Assuming
    The right conjunct: "(SUC m) + n = SUC(m + n)"
    The left conjunct: "0 + n = n"
```

```
...
This establishes
(SUC m) + n = SUC(m + n), 0 + n = n |- (SUC m) + n = SUC(m + (0 + n))
This establishes
0 + n = n |- (SUC m) + n = SUC(m + (0 + n))
This establishes
|- (SUC m) + n = SUC(m + (0 + n))
This establishes
|- (SUC m) + n = SUC(m + (0 + n))
This completes the proof of the conjecture
>> example7:
   "(SUC m) + n = SUC(m + (0 + n))"
```

8.4.2 The Resolution Transformers

The resolution functions `IMP_RES_THEN` and `RES_THEN`, like the function `CONJUNCTS_THEN`, are implemented in such a way that the application of the tactics of the form `IMP_RES_THEN` *f th* and `RES_THEN` *f th* to a goal do not introduce any assumptions, explicit or implicit, into the resulting subgoal. For example:

```
th = |- !x. x < 1 ==> (x = 0)
#let g = ["x < 1";"y < 1"],"(x = 0) /\ (y = 0)";;
g = (["x < 1"; "y < 1"], "(x = 0) /\ (y = 0)") : goal
#let gl,p = IMP_RES_THEN SUBST1_TAC th g;;
gl = [(["x < 1"; "y < 1"], "(0 = 0) /\ (0 = 0)")] : goal list
p = - : proof
...
thm =                          |- (0 = 0) /\ (0 = 0)
thm' =           y = 0, x = 0 |- (0 = 0) /\ (0 = 0)
thm'' = !x. x < 1 ==> (x = 0) |- (0 = 0) /\ (0 = 0)
#print_all_thm(p[thm]);;
y < 1, x < 1 |- (x = 0) /\ (y = 0)
#print_all_thm(p[thm']);;
y = 0, x = 0, y < 1, x < 1 |- (x = 0) /\ (y = 0)
#print_all_thm(p[thm'']);;
!x. x < 1 ==> (x = 0), y < 1, x < 1 |- (x = 0) /\ (y = 0)
```

Therefore, the implementations of `NAMED_IMP_RES_THEN` and `NAMED_RES_THEN` should have the same behaviour as `IMP_RES_THEN` and `RES_THEN` with respect

to assumptions. The technique used to implement `CONJUNCTS_THEN` can be adapted here; a whole proof step, in which the subgoal does not change, is devoted to displaying the resolvents, and the applications of the function f are described in subsequent steps. Care must be taken in implementing `NAMED_IMP_RES_THEN` and `RES_THEN` that the resolvents are used singly by f in the same order as in the corresponding ordinary functions.

```
This is the proof of the conjecture
>> example10:
   "(x = 0) /\ (y = 0)"
   Assuming
    The fact1: "x < 1"
    The fact2: "y < 1"

>>>> We use the theorem
     |- !x. x < 1 ==> (x = 0)
     to derive the following consequences from the assumptions made thus far:
     x < 1 |- x = 0
     y < 1 |- y = 0
     These theorems are used in sequence.  We are showing:

>> "(x = 0) /\ (y = 0)"
   Assuming
    The fact1: "x < 1"
    The fact2: "y < 1"

>>>> We substitute according to the following equality:
     x < 1 |- x = 0.
     Thus, it is sufficient to prove:

>> "(0 = 0) /\ (y = 0)"
   Assuming
    The fact1: "x < 1"
    The fact2: "y < 1"
   Assuming implicitly
    The hypothesis of the equality: "x < 1"

>>>> We substitute according to the following equality:
     y < 1 |- y = 0.
     Thus, it is sufficient to prove:

>> "(0 = 0) /\ (0 = 0)"
   Assuming
    The fact1: "x < 1"
    The fact2: "y < 1"
   Assuming implicitly
    The hypothesis of the equality: "y < 1"
    The hypothesis of the equality: "x < 1"

...

This establishes

|- (0 = 0) /\ (0 = 0)

This establishes
```

```
y < 1 |- (0 = 0) /\ (y = 0)

This establishes

y < 1, x < 1 |- (x = 0) /\ (y = 0)

This establishes

y < 1, x < 1 |- (x = 0) /\ (y = 0)

This completes the proof of the conjecture
>> example10:
   "(x = 0) /\ (y = 0)"
   Assuming
    The fact1: "x < 1"
    The fact2: "y < 1"
```

This seems a reasonably clear account. The fact that an implicit assumption is generated for each resolvent (i.e. for each theorem passed to the substitution function – `x < 1`, for example, is generated for the resolvent `x = 0`) is a no more minor imperfection, as these terms must be hypotheses of the final theorem in any case. That is, these terms are implicit assumptions in the sense that whether of not they are hypotheses of the theorem achieving the final subgoal, they will be hypotheses of the theorem achieving the original goal.

To devote a separate step to the use of each resolvent might seem tedious, but this is in fact the unseen effect of applying the ordinary `IMP_RES_THEN` *f th*. It is not in general the case that the sequence of uses of the resolvent-based theorems can be expressed as a single use of a list of theorems. For example, while a sequence of substitutions (via `SUBST1_TAC`) *can* be expressed as a single use of substitution (via `SUBST_TAC`), the same is not true of the functions `\th. REWRITE_TAC [th]` and `REWRITE_TAC`.

The function `NAMED_RES_THEN` is handled in a similar way to `NAMED_IMP_RES_THEN`.

9 Strip Functions

The strip functions are examples of HOL tactics that do not correspond to single 'natural' proof steps; they are convenient tactics that do one of several simple steps, and are often repeated to do at once all such simple steps that possibly can be done. They are also examples of tactic whose implementations makes clever use of higher order functions (namely, the functions described in Chapter ...), and as a result are difficult to understand

immediately. Some of the issues raised by the effort to give an account of an application of the strip functions are:

- To what extent to decompose the complex step into primitive (natural) steps;
- To what extent to give the account in terms of the implementation;
- How to identify the subgoals produced (and their assumptions) so that no mystery remains about their origin or parts.

9.1 The Strip Transformer in HOL

The basic stripping tool in HOL is the strip function `STRIP_THM_THEN`. Given a function *ttac* from theorems to tactics, a theorem *th*, and a goal *g*, `STRIP_THM_THEN` inspects the top level structure the conclusion of *th* and chooses amongst the tactic transformers `CONJUNCTS_THEN`, `DISJ_CASES_THEN` and `CHOOSE_THEN`, for conclusions which are conjunctions, disjunctions or existential terms, respectively, at the top level (and it fails for other terms). (The three tactic transformers are explained in Chapter ...)

```
STRIP_THM_THEN = FIRST_TCL [CONJUNCTS_THEN; DISJ_CASES_THEN; CHOOSE_THEN]
```

where

```
FIRST_TCL [ttcl1;...;ttcln] =  ttcl1  ORELSE_TCL  ...  ORELSE_TCL  ttcln
```

where

```
(ttcl1: thm_tactical) ORELSE_TCL (ttcl2: thm_tactical) ttac th =
  (ttcl1 ttac th) ? (ttcl2 ttac th)
```

(meaning: the value of the `ttcl1 ttac th` unless that evaluation fails, in which case the value of `ttcl2 ttac th`). The appropriate tactic transformer is then applied to *ttac*; then the resulting function to *th*; and finally, the resulting tactic to *g*. This is illustated by the following schematic examples:

```
...
g = ([], "t")
th1 = |- p1 /\ p2
th2 = |- p1 \/ p2
th3 = |- ?x. P x
#STRIP_THM_THEN ASSUME_TAC th1 g;;
([(["p2"; "p1"], "t")], -) : subgoals
#STRIP_THM_THEN ASSUME_TAC th2 g;;
([(["p1"], "t"); (["p2"], "t")], -) : subgoals
#STRIP_THM_THEN ASSUME_TAC th3 g;;
([(["P x"], "t")], -) : subgoals
```

`STRIP_THM_THEN` underlies the first of the two main strip tactics in HOL: `STRIP_ASSUME_TAC` *th*.

9.2 Stripping and Assuming a Theorem in HOL

The tactic `STRIP_ASSUME_TAC` *th*, applied to a goal *g*, maps the theorem *th* to one or more sets of clauses (terms), and assumes each set of terms (in the fashion of `ASSUME_TAC`) in a separate subgoal. The term part of each of the subgoals is unchanged. Each set of clauses is a subset of the basic (lowest level) disjuncts, conjuncts and witness subterms of the original term (with separate subgoals being formed for disjuncts). The effect of `STRIP_ASSUME_TAC` is illustrated with schematic theorems and goal:

```
#let g = [],"t:bool";;
g = ([], "t") : (* list # term)
...
th1 = |- p1 /\ p2
th2 = |- (p1 \/ p2) /\ (p3 \/ p4)
th3 = |- (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)
#STRIP_ASSUME_TAC th1 g;;
([(["p2"; "p1"], "t")], -) : subgoals
#STRIP_ASSUME_TAC th2 g;;
([(["p3"; "p1"], "t");
  (["p4"; "p1"], "t");
  (["p3"; "p2"], "t");
  (["p4"; "p2"], "t")],
 -)
: subgoals
```

```
#STRIP_ASSUME_TAC th3 g;;
([(["x < 2"; "p3"; "p2"; "p1"], "t"); (["x < 2"; "p3"; "p2"], "t")], -)
: subgoals
```

In each case, the clauses added to each subgoal are not themselves conjunctions, disjunctions or existential terms. The first theorem is mapped to a single subgoal, with the two conjuncts as separate assumptions. The second theorem induces a four-way disjunctive split, where the four subgoals have two clauses (disjuncts) each. The third would have eight subgoals, but two of these of these are solved internally because they are inconsistent, and two more because they are trivially true (i.e. they include the term `t` itself as an assumption). The two internal solutions preclude further case analysis, so that only six cases are actually generated. Of the two remaining subgoals, the second can be simplified to omit mention of the tautologous clause (`T`) and so includes only three clauses as assumptions. Both subgoals include the witness term `p2`.

`STRIP_ASSUME_TAC` is implemented by repeated use of `STRIP_ASSUME_THEN` and a version of `ASSUME_TAC`:

```
STRIP_ASSUME_TAC = (REPEAT_TCL STRIP_THM_THEN) CHECK_ASSUME_TAC
```

where

```
REPEAT_TCL (ttcl: thm_tactical) ttac th =
  ((ttcl  THEN_TCL  (REPEAT_TCL ttcl))  ORELSE_TCL  I) ttac th
```

and

```
(ttcl1: thm_tactical) THEN_TCL (ttcl2: thm_tactical) ttac = ttcl1 (ttcl2 ttac)
```

Rather than assuming the final clauses via `ASSUME_TAC`, `STRIP_ASSUME_TAC` uses the more selective function (`CHECK_ASSUME_TAC`) which notices and solves contradictions (via `CONTR_TAC`), and solutions (via `ACCEPT_TAC`). This introduces the possibility, therefore, of `STRIP_ASSUME_TAC` solving a goal. (`CHECK_ASSUME_TAC` also declines to add tautologous clauses as assumptions.)
To summarize:

```
STRIP_ASSUME_TAC th g
```

is

```
(REPEAT_TCL STRIP_THM_THEN) CHECK_ASSUME_TAC th g
```

which is

```
(REPEAT_TCL (FIRST_TCL [CONJUNCTS_THEN; DISJ_CASES_THEN; CHOOSE_THEN]))
  CHECK_ASSUME_TAC th g
```

which in turn is

```
(((FIRST_TCL [CONJUNCTS_THEN; DISJ_CASES_THEN; CHOOSE_THEN])  THEN_TCL
  (REPEAT_TCL ((FIRST_TCL [CONJUNCTS_THEN; DISJ_CASES_THEN; CHOOSE_THEN])))) ORELSE_T
I) CHECK_ASSUME_TAC th g
```

In the case of `th2` and `g`, above, for example, the ultimate 'chain' of theorem transformers contains two elements: `CONJUNCTS_THEN THEN_TCL DISJ_CASES_THEN`:

```
#CONJUNCTS_THEN (DISJ_CASES_THEN CHECK_ASSUME_TAC);;
- : thm_tactic

#(CONJUNCTS_THEN THEN_TCL DISJ_CASES_THEN) CHECK_ASSUME_TAC th2 g;;
([(["p3"; "p1"], "t");
  (["p4"; "p1"], "t");
  (["p3"; "p2"], "t");
  (["p4"; "p2"], "t")],
 -)
```

In general, `REPEAT_TCL STRIP_THM_THEN` results in a chain of functions f_1,...,f_n of type `thm_tactical` such that then `STRIP_ASSUME_TAC` is equal to $f_1(f_2(....(f_n$ `CHECK_ASSUME_TAC`$)...))$.

`STRIP_ASSUME_TAC` supports the two second of the two main strip tactics in HOL: `STRIP_TAC` th.

9.3 The Strip Tactic in HOL

The other main stripping tactic in HOL is `STRIP_TAC`, which performs one syntactic layer of stripping on a given goal. On goals whose terms are universally quantified, `STRIP_TAC` specifies to a variant of the quantified variable. On goals whose terms are conjunctions, it produces a pair of separate subgoals. The other possibility, aside from failure, is that the term is an implication, in which case the antecedent is taken apart into sets of clauses (by `STRIP_ASSUME_TAC`), and each set is assumed in a separate subgoal (whose term is the consequent of the implication). That is,

```
STRIP_TAC = STRIP_GOAL_THEN STRIP_ASSUME_TAC
```

where

```
STRIP_GOAL_THEN ttac =  FIRST [GEN_TAC; CONJ_TAC; DISCH_THEN ttac]
```

`STRIP_TAC` inherits from `STRIP_ASSUME_TAC` the ability to solve certain goals. Also, as is usual in HOL, a term of the form `~`t is regarded as being `t ==> F` so that `STRIP_TAC` approaches the proof of `~`t as a proof by contradiction.

`STRIP_TAC` is illustrated by adapting the theorems used above to illustrate `STRIP_ASSUME_TAC` – the antecedents are decomposed into disjuncts, conjuncts and witness terms:

```
g1 = ([], "p1 /\ p2 ==> t")
g2 = ([], "(p1 \/ p2) /\ (p3 \/ p4) ==> t")
g3 = ([], "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) ==> t")
#STRIP_TAC g1;;
([(["p2"; "p1"], "t")], -) : subgoals
#STRIP_TAC g2;;
([(["p3"; "p1"], "t");
  (["p4"; "p1"], "t");
  (["p3"; "p2"], "t");
  (["p4"; "p2"], "t")],
 -)
: subgoals
#STRIP_TAC g3;;
([(["x < 2"; "p3"; "p2"; "p1"], "t"); (["x < 2"; "p3"; "p2"], "t")], -)
: subgoals
```

Because of the inner *repeat* construct, an indefinite number of subgoals can result from an application of `STRIP_TAC`. That is, there may be any number of disjunctive splits, and of the subgoals generated, some may be solved.

9.4 Accounting for The Strip Tactic

One method of implementing `NAMED_STRIP_TAC`, to supply an account of the stripping process applied to a named goal, is to regard stripping as a compound proof step *not* to be accounted for as a single proof step. This is achieved by implementing `NAMED_STRIP_TAC` in parallel with HOL's `STRIP_TAC`, based on (likewise parallel) implementations of `NAMED_STRIP_GOAL_THEN`,

NAMED_STRIP_ASSUME_TAC, NAMED_STRIP_THM_THEN, NAMED_REPEAT_TCL, and so on. By this method, the job of constructing the account of the stripping tactic is handed over to the functions NAMED_CONJUNCTS_THEN and so on, giving, in the end, a full account of the processing of the goal, with each step in the process explained as a separate proof step.

A second method of implementing NAMED_STRIP_TAC is to gather and process the results of applying NAMED_STRIP_TAC. This gives an account of stripping as a single proof step. (The results of applying HOL's STRIP_TAC, to the corresponding ordinary goal – in the style of many other named tactics' implementations – does not give enough information to construct a useful account.)

We explain both methods, and leave the choice to be decided according to particular needs.

9.4.1 The Implementation-Based Account

Once all of the basic function are implemented for named goals, the tactic NAMED_STRIP_TAC is easy to implement in parallel with the HOL implementation. We consider three corresponding named goals:

```
ng1 = mk_named_goal(‘example1‘, [], "p1 /\ p2 ==> t")
ng2 = mk_named_goal(‘example2‘, [], "(p1 \/ p2) /\ (p3 \/ p4) ==> t")
ng3 =
mk_named_goal(‘example3‘,
              [],
              "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) ==> t")
```

To these we apply the version of NAMED_STRIP_TAC implemented in parallel with HOL's STRIP_TAC. In the first example, applying NAMED_STRIP_TAC to ng1 gives one subgoal:

```
>> "t"
   Assuming
    The right conjunct: "p2"
    The left conjunct: "p1"
   Assuming implicitly
    The antecedent: "p1 /\ p2"
```

The justification is constructed, as for HOL's STRIP_TAC, from the justifications of the constituent functions when the tactic is applied. Given an account of the subgoal, the justification returns an account of the whole stripping step:

```
This is the proof of the conjecture
>> example1:
   "p1 /\ p2 ==> t"

>>>> It is sufficient to prove:

>> "t"
   Assuming
    The antecedent: "p1 /\ p2"

>>>> We use the two separate theorems implied by the assumption
     p1 /\ p2 |- p1 /\ p2.
     The two theorems are used in sequence.  We are showing:

>> "t"
   Assuming implicitly
    The antecedent: "p1 /\ p2"

>>>> We use the fact that
     p1 /\ p2 |- p1.

     It is sufficient to prove:

>> "t"
   Assuming
    The left conjunct: "p1"
   Assuming implicitly
    The antecedent: "p1 /\ p2"

>>>> We use the fact that
     p1 /\ p2 |- p2.
     It is sufficient to prove:

>> "t"
   Assuming
    The right conjunct: "p2"
    The left conjunct: "p1"
   Assuming implicitly
    The antecedent: "p1 /\ p2"

...

This establishes

p1, p2 |- t

This establishes

p1, p1 /\ p2 |- t

This establishes

p1 /\ p2 |- t

This establishes

p1 /\ p2 |- t

This establishes

|- p1 /\ p2 ==> t
```

```
This completes the proof of the conjecture
>> example1:
   "p1 /\ p2 ==> t"
```

The account is straightforward; its second proof step is the one devoted by `CONJUNCTS_THEN` to explaining the conjunctive split of the antecedent assumption. The subgoal produced by this step is unchanged from the previous subgoal except for 'disappearance' of the (no longer needed) antecedent assumption at that point. The last subgoal shown has the antecedent of the original implication entirely taken apart, as a result of the steps determined by applying `NAMED_STRIP_TAC` to `ng1`.

When the chain of functions determined by applying `NAMED_STRIP_TAC` to a given goal is longer, and especially when it involves case splits (as it would in the second example), the account in the present style becomes more tedious and confusing. It is confusing, in particular, because there is a sequence of binary case splits to be presented, and the resulting cases are repeatedly labelled as the `left disjunct case` or the `right disjunct case`. The actual subgoal being considered at certain points in the presentation can be identified only via the convention that in printing a subgoal-proof tree in depth-first fashion, the next (awaiting) subgoal is re-printed immediately after a leaf has been printed.

Despite the inconveniences, it still sometimes the case that the account desired is the one that lays out all the stages of the stripping process. For example, the clearest explanation is produced for the third case by this method. Here, as mentioned earlier, there are two subgoals produced out of the six generated internally. These are:

```
>> left disjunct case:
   "t"
   Assuming
    The witness hypothesis: "x < 2"
    The left disjunct: "p3"
    The left disjunct: "p2"
    The left disjunct: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"
```

and

```
>> left disjunct case:
   "t"
```

```
   Assuming
    The witness hypothesis: "x < 2"
    The left disjunct: "p3"
    The left disjunct: "p2"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"
```

In the lengthy account produced by applying the justification, however, all six cases are displayed, and it is explained clearly how the four internal cases are solved (this information being provided by the named tactics that ultimately solve the internal goals). In contrast, it is not clear in HOL itself (see ...) how many cases were actually generated, nor of these, which were solved, and how.

```
This is the proof of the conjecture
>> example3:
   "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) ==> t"

>>>> It is sufficient to prove:

>> "t"
   Assuming
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

>>>> We use the two separate theorems implied by the assumption
     (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)
     |- (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2).
     The two theorems are used in sequence.  We are showing:

>> "t"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

>>>> We consider the two cases suggested by the fact
     (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) |- p1 \/ T

>> left disjunct case:
   "t"
   Assuming
    The left disjunct: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

>> right disjunct case:
   "t"
   Assuming
    The right disjunct: "T"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

The proof of the
>> left disjunct case:
```

```
   "t"
   Assuming
    The left disjunct: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

is as follows:
>>>> We use the two separate theorems implied by the fact
     (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)
     |- (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2).
     The two theorems are used in sequence.  We are showing:

>> "t"
   Assuming
    The left disjunct: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

>>>> We consider the two cases suggested by the fact
     (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) |- p2 \/ F

>> left disjunct case:
   "t"
   Assuming
    The left disjunct: "p2"
    The left disjunct: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

>> right disjunct case:
   "t"
   Assuming
    The right disjunct: "F"
    The left disjunct: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

The proof of the
>> left disjunct case:
   "t"
   Assuming
    The left disjunct: "p2"
    The left disjunct: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

is as follows:
>>>> We use the two separate theorems implied by the fact
     (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)
     |- (p3 \/ t) /\ (?x. x < 2).
     The two theorems are used in sequence.  We are showing:

>> "t"
   Assuming
    The left disjunct: "p2"
    The left disjunct: "p1"
```

```
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

>>>> We consider the two cases suggested by the fact
     (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) |- p3 \/ t

>> left disjunct case:
   "t"
   Assuming
    The left disjunct: "p3"
    The left disjunct: "p2"
    The left disjunct: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

>> right disjunct case:
   "t"
   Assuming
    The right disjunct: "t"
    The left disjunct: "p2"
    The left disjunct: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

The proof of the
>> left disjunct case:
   "t"
   Assuming
    The left disjunct: "p3"
    The left disjunct: "p2"
    The left disjunct: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

is as follows:
>>>> Using the term "x"
     as a witness to the fact
     (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) |- ?x. x < 2
     it is sufficient to prove:

>> "t"
   Assuming
    The witness hypothesis: "x < 2"
    The left disjunct: "p3"
    The left disjunct: "p2"
    The left disjunct: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

...

This establishes

x < 2, p1, p2, p3 |- t

This establishes
```

```
p3, (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2), p1, p2 |- t

The proof of the
>> right disjunct case:
   "t"
   Assuming
    The right disjunct: "t"
    The left disjunct: "p2"
    The left disjunct: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

is as follows:
>>>> The theorem
     t |- t
     is proposed to satisfy this.

This establishes

t |- t

This establishes

(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2), p1, p2 |- t

This establishes

p2, (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2), p1 |- t

The proof of the
>> right disjunct case:
   "t"
   Assuming
    The right disjunct: "F"
    The left disjunct: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

is as follows:
>>>> This follows vacuously (by contradiction) from the theorem

     F |- F

This establishes

F |- t

This establishes

(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2), p1 |- t

This establishes

p1, (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) |- t

The proof of the
>> right disjunct case:
   "t"
   Assuming
    The right disjunct: "T"
   Assuming implicitly
```

```
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

is as follows:
>>>> It is sufficient to prove:

>> "t"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

>>>> We use the two separate theorems implied by the fact
     (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)
     |- (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2).
     The two theorems are used in sequence.  We are showing:

>> "t"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

>>>> We consider the two cases suggested by the fact
     (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) |- p2 \/ F

>> left disjunct case:
   "t"
   Assuming
    The left disjunct: "p2"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

>> right disjunct case:
   "t"
   Assuming
    The right disjunct: "F"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

The proof of the
>> left disjunct case:
   "t"
   Assuming
    The left disjunct: "p2"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

is as follows:
>>>> We use the two separate theorems implied by the fact
     (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)
     |- (p3 \/ t) /\ (?x. x < 2).
     The two theorems are used in sequence.  We are showing:

>> "t"
   Assuming
    The left disjunct: "p2"
```

```
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

>>>> We consider the two cases suggested by the fact
     (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) |- p3 \/ t

>> left disjunct case:
   "t"
   Assuming
    The left disjunct: "p3"
    The left disjunct: "p2"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

>> right disjunct case:
   "t"
   Assuming
    The right disjunct: "t"
    The left disjunct: "p2"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

The proof of the
>> left disjunct case:
   "t"
   Assuming
    The left disjunct: "p3"
    The left disjunct: "p2"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

is as follows:
>>>> Using the term "x"
     as a witness to the fact
     (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) |- ?x. x < 2
     it is sufficient to prove:

>> "t"
   Assuming
    The witness hypothesis: "x < 2"
    The left disjunct: "p3"
    The left disjunct: "p2"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

...

This establishes

x < 2, p2, p3 |- t

This establishes
```

```
p3, (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2), p2 |- t

The proof of the
>> right disjunct case:
   "t"
   Assuming
    The right disjunct: "t"
    The left disjunct: "p2"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

is as follows:
>>>> The theorem
     t |- t
     is proposed to satisfy this.

This establishes

t |- t

This establishes

(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2), p2 |- t

This establishes

p2, (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) |- t

The proof of the
>> right disjunct case:
   "t"
   Assuming
    The right disjunct: "F"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

is as follows:
>>>> This follows vacuously (by contradiction) from the theorem

     F |- F

This establishes

F |- t

This establishes

(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) |- t

...

This establishes

|- (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) ==> t

This completes the proof of the conjecture
>> example3:
   "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) ==> t"
```

9.4.2 The Primitive Account

It may be the case that the explanation of the stripping process is *not* wanted, as above, in terms of the entire chain of steps, including the subgoals solved internally and the methods used – but simply in one unit strip step. If so, the strip function could not be implemented as above, in parallel with HOL's implementation.

Neither can it be implemented directly in an analogous way to many other tactics – by gathering and organizing the results of applying HOL's `STRIP_TAC` to the corresponding ordinary goal; this method does not give an adequate account because the results of `STRIP_TAC` in themselves afford no means of identifying the subgoals (and parts of subgoals) resulting from the stripping process.

Instead, the one-step function (`NAMED_PRIM_STRIP_TAC`, for 'primitive strip tactic') is implemented indirectly by applying the full-account version (`NAMED_STRIP_TAC`) to the goal and then processing those results into a single account. `NAMED_STRIP_TAC` gives enough information – via its constituent functions `NAMED_CONJUNCTS_THEN` and so on – to be able to identify the results in a meaningful way for accounting purposes.

The processing that is required on the results of applying `NAMED_STRIP_TAC` is quite elaborate. First, some simple processing greatly improve the account:

- Provision has to be made for the goal being completely solved, as that outcome is presented differently than a set of subgoals;
- It has to be noticed if the original goal is a negated term, so that the proof can be presented as a proof by contradiction;
- The term parameters of any applications of `NAMED_GEN_TAC` should be recorded; even though an individual generalization step is not going to be reported, this information may be required.

The more complex processing relates to the fact, observed earlier, that a single application of `STRIP_TAC` to an implicative goal can give rise to an indefinite number of subgoals, through a sequence of disjunctive splits of the antecedent, and through internal solutions. Subgoals arising in this way will always be identified (via `NAMED_STRIP_TAC`) as `left disjunct case` or `right disjunct case`. The final set of subgoals arising in this way can be recast by `NAMED_PRIM_STRIP_TAC` as a numbered sequence of disjunctive cases.

Withing each subgoal produced by `NAMED_STRIP_TAC` on an implicative goal, there may be various clauses (arising from the antecedent) which are identified as wintness hypotheses, left or right disjuncts, or left or right conjuncts. From these labels, the conjuncts' and disjuncts' names can be reorganized in numbered sequences.

For example, in the third case, it was mentioned earlier that the two visible subgoals (to be solved) were

```
>> left disjunct case:
   "t"
   Assuming
    The witness hypothesis: "x < 2"
    The left disjunct: "p3"
    The left disjunct: "p2"
    The left disjunct: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"
```

and

```
>> left disjunct case:
   "t"
   Assuming
    The witness hypothesis: "x < 2"
    The left disjunct: "p3"
    The left disjunct: "p2"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"
```

These can be recast and printed, respectively, as

```
>> disjunctive case 1 of 2:
   "t"
   Assuming
    The witness hypothesis: "x < 2"
    The disjunct 3: "p3"
    The disjunct 2: "p2"
    The disjunct 1: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"
```

and

```
>> disjunctive case 2 of 2:
   "t"
   Assuming
    The witness hypothesis: "x < 2"
```

```
    The disjunct 2: "p3"
    The disjunct 1: "p2"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"
```

The primitive account of the stripping step is then:

```
This is the proof of the conjecture
>> example3:
   "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) ==> t"

>>>> It is sufficient to prove the following:

>> disjunctive case 1 of 2:
   "t"
   Assuming
    The witness hypothesis: "x < 2"
    The disjunct 3: "p3"
    The disjunct 2: "p2"
    The disjunct 1: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

>> disjunctive case 2 of 2:
   "t"
   Assuming
    The witness hypothesis: "x < 2"
    The disjunct 2: "p3"
    The disjunct 1: "p2"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

The proof of the
>> disjunctive case 1 of 2:
   "t"
   Assuming
    The witness hypothesis: "x < 2"
    The disjunct 3: "p3"
    The disjunct 2: "p2"
    The disjunct 1: "p1"
   Assuming implicitly
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"

is as follows:
...

This establishes

p1, p2, p3, x < 2 |- t

The proof of the
>> disjunctive case 2 of 2:
   "t"
```

```
   Assuming
    The witness hypothesis: "x < 2"
    The disjunct 2: "p3"
    The disjunct 1: "p2"
   Assuming implicitly
    The right disjunct: "T"
    The antecedent: "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2)"
is as follows:
...
This establishes
p2, p3, x < 2 |- t
This establishes
|- (p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) ==> t
This completes the proof of the conjecture
>> example3:
   "(p1 \/ T) /\ (p2 \/ F) /\ (p3 \/ t) /\ (?x. x < 2) ==> t"
```

This account of applying `NAMED_PRIM_STRIP_TAC` does not explain the generation and solution of the four internal subgoals, but it does mirror the tactic `STRIP_TAC`, which takes apart the antecedent of an implicative goal and deals with the resulting clauses in a single proof step.

`NAMED_PRIM_STRIP_TAC` is implemented as an elaboration of the more basic `NAMED_STRIP_TAC`; it gives similar subgoals (the same with some renaming), but a different account. That is, `NAMED_PRIM_STRIP_TAC` computes the subgoals and justification (`p`, say) given by `NAMED_STRIP_TAC`, but then uses `p` to construct is own account. Its own account simply maps a given list of sub-accounts to an account (i.e. a node) with a name of its own, containing the given list of sub-accounts, the list of (processed) subgoals, and the theorem component of the account got by applying `p` to the list of sub-accounts. In this way, the theorem achieved is the only component of the long account (the account of `NAMED_STRIP_TAC`) that appears explicitly in the new account (the account of `NAMED_PRIM_STRIP_TAC`), although the same actual inferences are generated in both cases.

In a similar way, other patterns of inference also could be implemented to give one-step accounts. One simple instance of this would be a tactic to apply and account for `NAMED_PRIM_STRIP_TAC` repeatedly, in one step; this would be useful since `REPEAT STRIP_TAC` is a very commonly used beginning to proofs.

This idea forms the basis of a method for compacting long and excessively detailed accounts. Deciding which further patterns of inference could be presented coherently by being compacted into unit steps is a matter for future research.

10 Transforming Proof Accounts

Once the subgoal-proof tree has been extracted from the performance of a HOL proof, it can, in theory, be presented in a variety of ways – though just one style of presentation has been implemented to date. A further extension, however, is to transform the subgoal-proof tree itself before it is printed. This would be done in the interest of producing a clearer or more elegant proof, removing unnecessary proof steps, and so on. Such transformations would be based on a belief that the proof – in the sense of the sequence of inference steps corresponding to the subgoal-roof tree – were either preserved or were transformed in a validity-preserving way by the transformation of the tree[18]. This belief would be supported by a 'meta-argument' about the transformation rather than a re-derivation of the proof in the logic; that is, the correspondence of the new tree to a proof would be informal.

To date, two particular kinds of transformations have been implemented, to test this idea. Under the first transformation, uninterrupted sequences of generalization steps are compacted into a single, multiple generalization step (and the subgoal-proof tree reassembled accordingly). Under the second, steps which have no effect on a goal are removed and the remaining tree spliced together appropriately.

The following printed account results from a repeated application of `NAMED_STRIP_TAC` to the goal shown:

```
This is the proof of the conjecture
>> example:
   "!x y z. x < y /\ y < z ==> x < z"

>>>> Consider an arbitrary "x":
     We show:
```

[18]The subgoal-proof tree as defined does not include the inference sequence, but just the subset consisting of the theorems achieving the subgoals. These are produced, when the proof is performed, by computing the inference sequences in full; that is the sense in which there is a correspondence.

```
>> "!y z. x < y /\ y < z ==> x < z"
>>>> Consider an arbitrary "y":
     We show:
>> "!z. x < y /\ y < z ==> x < z"
>>>> Consider an arbitrary "z":
     We show:
>> "x < y /\ y < z ==> x < z"
...
```

When the subgoal-proof tree which underlies this account is transformed in the first way, a new tree is produced. The new tree is printed as follows:

```
This is the proof of the conjecture
>> example:
   "!x y z. x < y /\ y < z ==> x < z"
>>>> Considering arbitrary "x", "y", "z",
     we show:
>> "x < y /\ y < z ==> x < z"
...
```

This transformation is achieved by collecting from the original tree all uninterrupted sequences of steps which are equivalent in effect to generalizations and then representing each sequence as a single node in a new tree. The single node is conceived as representing a multiple generalization tactic – a tactic equivalent in its effect to an application of `REPEAT GEN_TAC` but considered as a single proof step. Steps equivalent in effect to generalizations might have been generated by application of `GEN_TAC`, or might have been generated indirectly, e.g. via application of `STRIP_TAC`, provided that indirect generalizations manage to record the variable in question in the same way that `GEN_TAC` does.

That is, an account of the form

```
mk_node((`NAMED_GEN_TAC`, ["x"], []),
        [mk_node((`NAMED_GEN_TAC`, ["y"], []),
                 [mk_node((`NAMED_GEN_TAC`, ["z"], []),
                          ... ,
                          [mk_named_goal(`example`,
                                         [],
                                         "x < y /\ y < z ==> x < z")],
                          |- !z. x < y /\ y < z ==> x < z)] ,
                 [mk_named_goal(`example`,
```

```
                                []),
                                "!z. x < y /\ y < z ==> x < z")],
                        |- !y z. x < y /\ y < z ==> x < z )],
        [mk_named_goal(`example`, [], "!y z. x < y /\ y < z ==> x < z")],
        |- !x y z. x < y /\ y < z ==> x < z )
```

becomes an account of the form

```
mk_node((`MULTI_NAMED_GEN_TAC`, ["x"; "y"; "z"], []),
        ... ,
        [mk_named_goal(`example`, [], "x < y /\ y < z ==> x < z")],
        |- !x y z. x < y /\ y < z ==> x < z)
```

where `MULTI_NAMED_GEN_TAC` is a new kind of node (suggesting a hypothetical new tactic) with its own printing convention. (The node and its printing format must of course be known to the printing functions in advance.)

Redundant proof steps arise for a variety of reasons; for example, the use of tactics which never fail (e.g. rewriting), or linear tactics which advance one branch of a proof but which neither fail nor have any effect on the another branch. For example, if the goal of the previous example is attacked by applying to it the (rather odd) tactic

```
NAMED_REWRITE_TAC [] THEN
NAMED_STRIP_TAC THEN
NAMED_REWRITE_TAC []
```

so that only the `STRIP_TAC` advances the proof, the following account is printed:

```
This is the proof of the conjecture
>> example:
   "!x y z. x < y /\ y < z ==> x < z"

>>>> Using basic tautologies, it is sufficient to prove:

>> "!x y z. x < y /\ y < z ==> x < z"

>>>> Consider an arbitrary "x":
     We show:

>> "!y z. x < y /\ y < z ==> x < z"

>>>> Using basic tautologies, it is sufficient to prove:

>> "!y z. x < y /\ y < z ==> x < z"

...
```

Under the second transformation, the redundant steps are removed from the tree, and the resulting tree is printed as follows:

```
This is the proof of the conjecture
>> example:
   "!x y z. x < y /\ y < z ==> x < z"
>>>> Consider an arbitrary "x":
     We show:
>> "!y z. x < y /\ y < z ==> x < z"
...
```

This transformation is achieved by searching for nodes which have exactly one direct descendent node, and for which the subgoal is the same as the goal[19]. Where there is a single unchanged subgoal, the transformation involves removing the subgoal node from the tree and splicing up the rest of the tree accordingly. The transformation applies recursively throughout tree.

In the example above, the original account has the form

```
mk_node((`NAMED_REWRITE_TAC`, [], []),
        [mk_node((`NAMED_GEN_TAC`, ["x"], []),
                 [mk_node((`NAMED_REWRITE_TAC`, [], []),
                          ... ,
                          [mk_named_goal(`example`,
                                         [],
                                         "!y z. x < y /\ y < z ==> x < z")],
                          |- !y z. x < y /\ y < z ==> x < z)],
                 [mk_named_goal(`example`,
                                [],
                                "!y z. x < y /\ y < z ==> x < z")],
                 |- !x y z. x < y /\ y < z ==> x < z)],
        [mk_named_goal(`example`,
                       [],
                       "!x y z. x < y /\ y < z ==> x < z")],
        |- !x y z. x < y /\ y < z ==> x < z)
```

while the transformed tree has the form

```
mk_node((`NAMED_GEN_TAC`, ["x"], []),
        ...
        [mk_named_goal(`example`, [], "!y z. x < y /\ y < z ==> x < z")],
        |- !x y z. x < y /\ y < z ==> x < z)
```

Both of the transformations can be done in a single combined transformation which applies repeatedly until neither tranformation can assist.

[19] 'The same' is taken in the first instance to mean identical except for the goals' names, though more subtlety may be called for in treating implicit assumptions, etc.

Another use of such transformations might be to print implicit assumption more selectively (e.g. where they are duplicated), or not at all (in contexts where they are not of interest).

Some elaborations along these lines are mentioned in Chapter ... on future research ideas. The two described here are very simple transformations, but the idea could be extended to more sophisticated transformations which resulted in accounts which are preferred for some purpose. It is worth stressing again, however, that transforming and re-printing the internal respresentation of a proof does not entail re-proving anything. The transformed trees may indeed fail to represent valid proofs – despite any informal arguments that they do, the trees may no longer correspond to valid proofs.

To achieve a direct correspondence, it might be possible, as a side effect of transforming the tree, in some cases, to derive automatically the new tactic that corresponds to the transformed tree, and then to try to apply that tactic to the original goal. If this worked, it would produce the new (genuine) tree directly. Clearly, this makes no sense where a hypothetical tactic is suggested (such as `MULTI_NAMED_GEN_TAC`, mentioned earlier), but it should be possible, for example, for the second kind of transformation. However, this idea is mere speculation at present.

11 Future Research

We mention briefly in this Chapter some extensions of the account facility which we hope to make in future work. These are grouped as practical and theoretical extensions.
Some theoretical extensions are as follows:

- The idea of transforming trees before printing (Chapter ...) could be extended to more sophisticated transformations. One sort of transformation which might be helpful would be the selective presentation of proof steps, with the ellipsis or omission of other steps. For example, it might be desired, particularly in long proofs, to produce accounts consisting only of the major or important proof steps. The full accounts shown in this paper are probably too long and detailed for some purposes. Part of the research would be to decide which steps in which contexts are 'important'.

- We also mentioned (in Chapter ...) the idea of extracting from the transformation process enough information to be able to construct the transformed tactic, at least in certain cases. A particular application of this would be to rephrase HOL tactics in some desired style. For example, once the subgoal-proof tree is known, the compound tactic which produced the tree might could be rephrased to be more linear (so that separate branches are generated by one flat sequence of tactics) or less linear (so that selective sequencing – `THENL`, for instance – were used where branching occurs). This would be useful where such uniformity of style is desired.

- At present, it is required that a proof be successfully completed in HOL before an account can be generated – by re-performing the proof in a different mode. It might also be useful to be able to work piecewise and interactively; that is, to generate an account of one step within a proof. This would be useful, for example, for understanding mysterious single steps in completed proofs, or for assessing the effect of difficult steps in a proof in progress. An interactive facility would involve changing the new ML types (Chapter ...) to some extent, since an account, as things stand, includes the achieving theorem associated with each node. However, the basic concepts should make some sort of interactive facility possible.

- In connection with the above point, another role of the account facility might be as a proof debugging aid. That is, where a proof fails, or proceeds on an unexpected course, the explanation of certain steps may be valuable in tracing the cause of the problem. Having access to the subgoal and its purported achieving theorem at a problem point may provide the key to understanding the failure. Here, any implicit assumptions (which will be accessible) may also shed light on the problem. Accounts seem particularly useful where a tactic implemented by a user directly in ML fails in some way.

- It would also be useful if the account facility could be integrated with another facility for explaining segments of forward proof. (A facility for explaining forward proofs is part of a currently proposed research grant.) If explanations of the interludes of forward proof which sometimes occur in goal-oriented proofs could be generated, it would be

possible to give more information within accounts as presented so far. For example, where a rewrite rule is derived by a sequence of forward inferences, the existing account facility would just report a rewriting event based on the theorem resulting from the forward inference. If the inference could itself be explained, the new theorem would not appear as if by magic, but would be accounted for meaningfully.

- In relation to the above point, one slightly unsatisfactory feature of the accounts produced currently for rewiting steps is that a rewriting step of a proof is reported based on *all* of the (potential) rewrites provided. In fact, it would be more informative to be told which rewrites were actually engaged and which were not, in each case. There appeared to be no simple, accurate way to do this within the accounting scheme presented. 'Named' tactics were generally implemented by elaborating on the results of the original tactics; original tactics were taken as 'black boxes'. Rewriting, in particular, has a complex and sensitive implementation in HOL, it seemed sensible to avoid trying to re-implement it accurately. It also seemed within the spirit to the current account package not to re-implement it. However, if there were already a way of tracing the actual steps of the rewriting process as part of a system for explaining forward proofs, this would make a valuable addition to the existing proof account facility for rewriting.

- It might be worth making a wider study of textbook-style proof presentations with the aim of improving the style of proof account printouts.

- The HOL package for introducing recursive data types and automatically generating induction rules for them was designed and implemented by Tom Melham (...). Derivation of induction rules follows from the definitions that characterize the new recursive data type. We have discussed numerical induction only in this paper (...), but it would be very desirable if, from any new recursive type definition, one could automatically generate the 'named' tactic which would produce the appropriate account. This seems in principle to be possible, but has not yet been studied carefully.

- It seems possible that the naming of assumptions in the new system of ML types needed for generating accounts may have other applications.

One obvious application is the accessing of assumptions by name rather than by position in the (arbitrary) order imposed by a particular HOL implementation. That is, if an induction hypothesis is identified by the string `‘induction hypothesis‘`, then one ought to be able to say something like ‘rewrite using the induction hypothesis as a rewrite rule’ rather than ‘rewrite using the third assumption (which I happen to believe is the induction hypothesis, at the moment)’. This would be a great convenience to the user, and moreover would produce much clearer accounts.

- It would be desirable to test many more examples of ML constructs which users employ in generating proofs in HOL, particularly the more complex ones. There is probably too much bias in examples constructed for the purpose.

Some practical extensions are as follows:

- The first project is to prepare a cleaner and more efficient implementation suitable for being released with the HOL system (along with suitable documentation). The facility should also be better interfaced to the HOL system, and easier to use. For example, one would like to switch into a mode in which accounts were generated (and switch out again, perhaps) without having to use new names for tactics (e.g. `NAMED_STRIP_TAC` for `STRIP_TAC`, etc).

- The existing accounts facility applies, of course, only to standard HOL tactics. For users who implement their own tactics (in ML rather than as combinations of standard functions), there is no way to produce accounts except by implementing directly the original tactics as named tactics. It might be possible to provide an interface for allowing users to accomplish this more easily. The interface could, for example, ask the user what to call the subgoals and any new assumptions, and so on, and then implement the original tactic in a uniform way.

- New printing styles should be tried; the one used in this paper is only a first attempt.

- A new package for managing goal-oriented proofs (i.e. a new subgoal package) has recently been implemented by Sara Kalvala (...). (This is

a standard part of the HOL 12 implementation.) This package involves an internal respresentation of the proof tree, and includes a means of extracting the text of a tactic from the interaction during which a proof is developed. It would be interesting to explore the relation of that package to the account facility, and any ways in which the two could be combined, or could benefit from each others' techniques and ideas.

- It was mentioned (Chapter ...) that the standard function `POP_ASSUM` causes a slight anomaly in that its justification does not 'replace' the lost assumption in a given achieving theorem. This was particularly apparent in tactics such as `POP_ASSUM(K ALL_TAC)`. One small future experiment would be to re-implement `POP_ASSUM` so that its justification *did* add the popped assumption to the incoming theorem, and to establish that this repair worked correctly with other functions. If so, the idea of implicit assumptions would become simpler. (This point relates to the discussion on pages ...).

12 Conclusions

The main purpose of the work described here has been to test the feasibility of extracting a conventional or 'natural' explanation of a proof from the process of performing the proof in HOL (in goal-oriented fashion). It was intended that this explanation be free of concepts specific to HOL or to mechanized theorem-proving, even where the HOL tactics used were specialized or obscure. The main questions were: could enough information be extracted from the application fo tactics to a goal to compose an explanation oif the proof? What was is the essential information? What is involved in presenting it in readable form?

So far, the ideas for assembling explanations seem to have worked well, and the explanations produced, at least for the basic tactics and tactic constructions seem reasonable. However, a great deal more experimentation with real proofs (and in particular with other users' proofs) is still required. We plan to pursue this in future. As mentioned in Chapter ..., the accounts produced at the moment are probably too detailed and exhaustive for some purposes, and it is planned also in future to experiment with ideas for con-

densing them. The particular style and layout used in this paper are only preliminary, and these may change with experience. At prsent, what we have is a basis for explaining proofs, and a framework in which to introduce refinements.

The main obstacle thus far to producing accounts was dealing with tactics formed by applying 'continuation' functionals to tactics. Though this is a flexible and convenient method for the HOL user, such constructs have the effect of performing some of the proof steps behind the scenes, and doing more than one major proof step at a time. The resulting leap is therefore difficult to explain. We have proposed one way of spelling out such steps (in Chapter ...) which seems to produce a comprehensible story. The method proposed may appear slightly unsatisfactory in that it reverses the direction of the HOL implementation, in which the higher order functionals (e.g. the continuations) are primary and the ordinary tactics are defined in their terms; the method for producing accounts in these cases takes the tactics as primary and the higher-order constructs defined in terms of *them*. However, there is no real reason to insist that the concepts and tools of the HOL user be determined by what happens to be the implementation of HOL. For example, the HOL system is normally taught by presenting simple tactics first, and tactic constructions later on (if at all).

A second, related obstacle (see Chapter ...) was the use of the set of current assumptions as a stack or array, in which the position of an assumption – which is again just an artefact of the HOL implementation – provides a means of accessing assumptions. This approach occasionally also involves the apparent 'dropping' of assumptions once they are 'used', partly as a means of controlling the size of the assumption set. Our analysis points to various conceptual problems in this style of proof, but as the method is now popular in the HOL community, it seemed necessary to provide a way of accounting for proof steps based on a stack or array of assumptions. We think that the method proposed in Chapter ... is quite satisfactory.

The means of overcoming both of the above obstacles, and to the problem of invalid proof steps as well (see Chapter ...), suggested the notion of *implicit assumptions*. That concept is introduced in Chapter ... . By making accessible all the assumptions which hold at a given stage in a goal-oriented proof, but which are not normally made explicit, several mysteries about HOL proofs can be cleared up. At the same time, *always* printing implicit assumptions creates a certain amount of clutter. Further work is planned on

how to decide exactly when printing implicit assumptions is useful.

13 References

14 Appendix

This appendix lists (i) the ML functions which work as they are under the new system of ML types (described in Chapter ...); (ii) the ML functions which have been re-implemented for the new system of types; and (iii) new functions which have been implemented for the new system of types. Each function is listed with its main appearance in the text.
The functions which do not require modification for HOL (Version 11) are:

```
THEN
THENL
MAP_EVERY
EVERY
FIRST
MAP_FIRST
NO_TAC
ORELSE
REPEAT
THENC
```

The functions which have been re-implemented are:

```
NAMED_GEN_TAC
NAMED_X_GEN_TAC
NAMED_INDUCT_TAC
NAMED_SUBST_TAC
NAMED_SUBST1_TAC
NAMED_BOOL_CASES_TAC
NAMED_COND_CASES_TAC
NAMED_SPEC_TAC
NAMED_ASSUME_TAC
NAMED_ACCEPT_TAC
NAMED_ASM_CASES_TAC
NAMED_CONJ_TAC
NAMED_LIST_INDUCT_TAC
NAMED_ALL_TAC
NAMED_EQ_TAC
NAMED_CONV_TAC
NAMED_EXISTS_TAC
NAMED_MP_TAC
NAMED_UNDISCH_TAC
NAMED_CONTR_TAC
NAMED_DISCARD_TAC
NAMED_MATCH_MP_TAC
NAMED_MATCH_ACCEPT_TAC
NAMED_SUBST_OCCS_TAC
NAMED_BETA_TAC
NAMED_REWRITE_TAC
NAMED_ASM_REWRITE_TAC
NAMED_PURE_REWRITE_TAC
NAMED_ONCE_REWRITE_TAC
NAMED_PURE_ASM_REWRITE_TAC
NAMED_PURE_ONCE_REWRITE_TAC
NAMED_ONCE_ASM_REWRITE_TAC
NAMED_PURE_ONCE_ASM_REWRITE_TAC
NAMED_DISCH_TAC
NAMED_DISCH_THEN
```

```
NAMED_DISJ_CASES_TAC
NAMED_DISJ_CASES_THEN2
NAMED_DISJ_CASES_THEN
NAMED_X_CHOOSE_TAC
NAMED_X_CHOOSE_THEN
NAMED_CHOOSE_TAC
NAMED_CHOOSE_THEN
NAMED_CONJ_ASSUME_TAC2
NAMED_CONJUNCTS_THEN2
NAMED_CONJUNCTS_THEN
NAMED_IMP_RES_TAC
NAMED_RES_TAC
NAMED_IMP_RES_ASSUME_TAC
NAMED_IMP_RES_THEN
NAMED_RES_ASSUME_TAC
NAMED_RES_THEN
$MY_THEN_TCL
$MY_ORELSE_TCL
MY_REPEAT_TCL
MY_ALL_THEN
MY_NO_THEN
MY_EVERY_TCL
MY_FIRST_TCL
NAMED_CHECK_ASSUME_TAC
NAMED_STRIP_THM_THEN
NAMED_STRIP_ASSUME_TAC
NAMED_STRIP_GOAL_THEN
NAMED_STRIP_TAC
NAMED_SUBST_ALL_TAC
NAMED_ASSUME_LIST_TAC
NAMED_ASSUM_LIST
NAMED_FIRST_ASSUM
NAMED_CHANGED_TAC
NAMED_REFL_TAC
NAMED_THEN_TCL
NAMED_ORELSE_TCL
NAMED_REPEAT_TCL
NAMED_EVERY_TCL
NAMED_FIRST_TCL
NAMED_ALL_THEN
NAMED_NO_THEN
```

The new functions which have been implemented are:

```
C_NAMED_ASSUME_TAC1
C_NAMED_ASSUME_TAC2
NAMED_POP_TRACE
NAMED_POP_TRACE'
NAMED_POP_TRACE''
NAMED_POP_TRACE'''
NAMED_POP_ASSUM
NAMED_POP_ASSUM'
NAMED_POP_TRACE_LIST
NAMED_POP_TRACE_LIST'
NAMED_POP_TRACE_LIST''
NAMED_POP_ASSUM_LIST
NAMED_BASIC_IMP_RES_TAC
NAMED_PRIM_STRIP_TAC
```