Automatically Synthesized Term Denotation Predicates: A Proof Aid

Paul E. Black* and Phillip J. Windley

Computer Science Department, Brigham Young University, Provo UT 84602, USA

Abstract. In goal-directed proofs, such as those used in HOL, tactics often must operate on one or more specific assumptions. But goals often have many assumptions. Currently there is no good way to select or denote assumptions in HOL88. Most mechanisms are sensitive to inconsequential changes or are difficult to use.
Denoting assumptions by filters (matching) makes it easier to maintain large proofs and reuse pieces. But writing the filter predicate can be time-consuming and distracting.
We describe an aid to proof building which synthesizes filter functions from terms. Given examples of terms which should and should not be matched, the function creates an ML predicate which can be used, for example, with `filter` or `FILTER_ASM_REWRITE_TAC`.
This paper reviews past discussions on denotation methods, the design and implementation of the filter synthesizer, applicable AI classification techniques, and possible application to more general term handling and recognition.

1 Introduction

Proofs take a lot of time to create. To make proofs more widely applicable, the time and expertise necessary to create a proof must be reduced. One approach is to reuse large portions of proof scripts (tactics) in similar proofs. To be able to reuse portions of proof scripts, minor differences in goals should require minimal changes to the proof script.

Changes arise because the theorem to be proved changes and because the HOL implementation changes. One area which has caused lots of possibly avoidable work is in denoting or choosing assumptions. Currently there is no good way to denote or choose assumptions in HOL88.

There are many ways to denote assumptions, but all have drawbacks.

- Denoting assumptions by their position in the assumption list is simple, but if other tactics are used or tactic implementation changes, the position of assumptions may change.
- Denoting assumptions by quotation works regardless of position, but fails if the goal changes even slightly.

* This work was sponsored by the National Science Foundation under NSF grant MIP–9412581

– Naming assumptions, then referring to them by name is robust. However it is not available in HOL88 and is just becoming available in HOL90.
– Denoting assumptions by filtering is insensitive to the order of assumptions and can be insensitive to changes in variable names and details of the theorem to be proved. However it is time-consuming and tedious to come up with appropriate filter predicates.

In Sect. 2 we review these approaches and others which have been suggested over the years in more detail. We compare their advantages and disadvantages. In Sect. 3 we present the design of the proof aid and its implementation. Section 4 presents alternative classification techniques and algorithms for constructing predicates. Finally we give some examples in Sect. 5, and Sect. 6 is our conclusions and ideas for future work.

2 Approaches to Denoting or Selecting Assumptions

Over the years the HOL community has suggested and discussed many approaches to denoting or selecting assumptions. Here we enumerate them and compare their strengths and weaknesses.

2.1 Denoting by Quotation

The HOL "handbook" [7] mentions the "general problem of denoting assumptions" and says

> The only straightforward way to denote them in the existing system is to supply their quoted text. Though adequate, this method may result in bulky ML expressions; and it may take some effort to present the text correctly (with necessary type information, etc.).

2.2 Denoting by Position

The book goes on to describe two other approaches: treating the assumption set as a stack and intercepting and manipulating results without them being added as assumptions.

The first approach uses *pop* operations to minimize the number of assumptions and denote the top assumption without explicit quotation. This is workable if the number of assumptions is small. The generalization of this approach, denoting assumptions by their position in the list, is sensitive to changes in tactic implementation or proofs.

2.3 Immediate Use

The second approach employs tacticals such as `DISJ_CASES_THEN` and `DISCH_THEN` to use results immediately without ever making them assumptions.

Ching-Tsun Chou [5] gave a detailed explanation of theorem continuations and their use. These techniques reduces the problem of denoting assumptions by reducing the number of assumptions and using some results directly. However proofs still have many assumptions at some points.

2.4 Reference by Position, Denoting by Quotation

David Shepherd [13] suggests a rather radical approach. The user refers to assumptions by their position in the assumption list, and a tactic recording system replaces the reference with a quotation of the assumption. The user then saves a "tactic script" for future use. The HOL system would require changes. Worse yet, when the proof is rerun after small changes, such as variable names, the quoted assumption would not match, the tactic would fail, and the script would require human maintenance.

Even more radical would be to have a system recording the tactics which are used and which tactic added which assumption. One refers to assumptions by position, and the tactic recording system modifies the step which added the assumption so it will save the assumption for use at this point. This would require even more extensive changes, and could be accomplished by naming (explained in Sect. 2.6).

2.5 Saving Terms and Denoting by Quotation

Paul Curzon [6] suggested saving assumptions in ML variables during the proof, then using them, e.g. via ASSUME, when needed. This is insensitive to changes of position or nature of term. The variable name could serve as documentation, too. The drawback is that some tactics don't return the results, although Chou's [5] theorem continuations could be adapted to the purpose.

2.6 Denoting by Name

Sara Kalvala [10] developed an extension of HOL90 in which terms and other structures can have labels (among other things). She also presented some specialized tactics which denote assumptions by label. Nuprl 4 allows users to label assumptions [12], too. Labeling assumptions is probably the best approach in the long run: proof scripts are insensitive to most changes and names can act as documentation or hints.

2.7 Denoting by Filter Predicate

John Harrison [9] suggested using a filter predicate, such as

```
FIRST_ASSUM(SUBST1_TAC o assert (curry $= "x:num" o lhs o concl))
```

This can be done in HOL88 and is less sensitive to changes in proof scripts than denotation by quotation or position. Jim Alves-Foss [2] pointed out that although proofs written with filters are insensitive to many changes, they may not be very readable. Denoting assumptions by quotation documents what assumption is used at a step.

Another drawback is that it is difficult to write filter functions. If the assumptions to be denoted are very different from all the others, a few selectors, such as `is_abs` or `\t.is_comb t & rator t = $+`, are sufficient. But a predicate of any significant depth will be a confusing jumble of selectors little better than the `car`'s, `cdr`'s, and `cadddr`'s sometimes seen in LISP. For example, here is a predicate used in Uinta verification:

```
let machine_pred thm =
    let tm = concl thm in
    (((rand o fst o dest_eq o snd o dest_forall) tm)
                              = "lmdr(t + 1):*wordn") ? false;;
```

Clearly the predicate is checking for `lmdr(t + 1)` at some specific place in the term, but it is not clear where. When the term changes, it will take some work to rewrite the predicate.

3 The Predicate Synthesizer

In this section we describe a new tool which synthesizes a predicate from examples of terms. We detail the design requirements and explain the implementation.

3.1 Requirements

The tool, `find_filter`, takes two lists: a list of terms and a list of indices of those terms which *must* be matched. Terms whose indices are not listed must *not* be matched. Terms which must match are called **positive examples** in Artificial Intelligence literature [14]. Terms which must *not* match are called **negative examples**. The synthesized function is returned as a string with full types and can be incorporated in the proof script for later use. The type of the tool is then `find_filter: term list -> int list -> string`, and the synthesized function has the type `: term -> bool`.

The synthesized function should be insensitive to minor changes in the proof. Therefore it should recognize general differences and similarities instead of specifics which are peripheral to the proof. As an example of what we *don't* want, the function `\t. t = pos1 \/ t = pos2 \/ . . .\/ t = posN` (where `pos1` through `posN` are the positive examples) always filters exactly (as long as no term is both a positive and negative example), but it is verbose and sensitive to any change in the positive terms.

The synthesized function must be fast since some proofs may filter assumptions hundreds or thousands of times. Therefore the function should be a series of predicates, such as

```
\t.is_forall t & vname (bndvar (rand t))=`t`
               & is_imp (body (rand t)) ? false
```

The tool is acceptable even if it takes some minutes to synthesize the filter function since the user only runs it occasionally, and then at human interaction speeds. The synthesized function should be somewhat readable, so the user can gauge how general the function actually is. This is not crucial since even if the proof changes enough to invalidate the function, the new assumptions can be added to the "training set" and the tool rerun to yield a more robust function.

3.2 Implementation Philosophy

The tool should compare the positive and negative example and synthesize not just *some* function which discriminates between the two sets, but a function which covers the most possibilities. That is, the function should continue to discriminate correctly in the future as far as possible. We would have to be clairvoyant to write a program which always chooses correctly, but how good can we do in practice?

Carbonell, et. al. [3] considers this problem as learning a classification from examples. The examples came from the external environment, that is, the examples are not specifically chosen or designed to teach the concept. Both positive and negative examples without "noise" are available, and the resultant classification must be correct for all examples. In learning the classification, the synthesizer should find features of terms which make up the concept "desired terms." The general approach we chose is to search a rule-version space [14].

Abstraction Level: One Possible Feature Set

Terms often have a general form in common, for instance, all positive example terms may be implications while none of the negative examples are. This suggests extracting features based on parts closest to the top, or root, of the abstract syntax tree or parse tree of a term. Consider the term `"!x y. x < 3 /\ y > 0"`. The most general recognition function of the term is `\(t:term).true`; all detail of the term has been abstracted away. Since there is no detail, we call it a level 0 abstraction; none of the parse tree of the term is used. The function which only includes one level of detail from the top, that is the level 1 abstraction from the top, is `\t.is_forall t`. It checks that the first level of the term matches, but nothing more. The level 2 top abstraction is

```
\t.is_forall t & is_conj (body (rand (body (rand t)))) ? false
```

A rigorous description more clearer phrased as testing for a match with a "term pattern." A term pattern is a term which may have specially named variables that match any (sub)term. (The special variables are named **STARn** reminiscent of the Kleene Star.) The level 0 abstraction of any term is just a **STAR** variable, i.e., a pattern which matches anything (the type is copied from the term). A level n abstraction, where $n > 0$, is as follows. The level n abstraction of a constant or variable is that constant or variable: an exact match. The level n abstraction of a λ-term is **mk_abs** of the level $n-1$ abstraction of the bound variable and the body.

The level n abstraction of a combination term is more complex. If the rator is a binder, such as $\forall$ or $\exists$, it is the level $n-1$ abstraction of the bound variable and body of the binder's abstraction:

```
mk_comb(
    rator t,
    mk_abs(
        abstraction (n-1) (bndvar (rand t)),
        abstraction (n-1) (body (rand t))
    )
)
```

If the rator is not a binder, it is the combination of the level n abstraction of the rator and the level $n-1$ abstraction of the rand. That is, if an operator is at level n, *all* its operands are at level $n+1$.

3.3 Implementation

The filter function is synthesized in several steps.

1. Find one or more term patterns such that all positive terms are matched by at least one pattern and the pattern do not match any negative terms.
   (a) Arbitrarily choose a positive example. Find the greatest abstraction (least level n, from Sect. 3.2 above) of it which matches all the positive examples and none of the negative examples.
   (b) Slightly generalize the abstraction if possible.
   (c) If no abstraction matches all the positives and none of the negatives, find a pattern for the first positive and a pattern for the rest of the positives separately.
2. Convert the term pattern(s) into an ML predicate which matches the same terms. The ML predicate is constructed as a string so it can be easily printed. The output can then be used as a filter function in a proof.

Obviously this limited exploration of possible term patterns may fail to find a set which works. If none of these work, `find_filter` ends with the message, no filter found.

Abstractions are slightly generalized if possible. A new pattern, with some penultimate node of the pattern tree replaced by a `STAR`, is formed and checked against the negatives. If it does not match any of the negatives, it is used. For example, suppose a term pattern is `STAR2 /\ STAR3 ==> STAR3 \/ STAR4`. The new patterns formed by slightly generalizing it are `STAR2 /\ STAR3 ==> STAR5` and `STAR6 ==> STAR3 \/ STAR4`. Both of these are slightly more general than the original pattern.

In some cases no single pattern matches all of the positives and none of the negatives. To handle these, the positives are split into two sets, the first positive and the rest. The code attempts to find patterns which match the first, but not the negative, and the rest, but not the negatives. The final predicate then checks for a match with either of these patterns. Clearly this could be enhanced to create as many patterns as needed to match all of the positives, but none of the negatives.

The final term pattern (or patterns) is converted to a string of appropriate ML code. For instance, a conjunction in the pattern becomes `is_conj` of the appropriate selection *and* (`&`) predicates to test both conjuncts, if needed. `STAR`s match anything, so no predicate is needed for them. Special characters are quoted so the text can be incorporated directly into proof scripts. We give examples of the synthesized functions in Sect. 5.

4 Alternative Algorithms and Classification Techniques

This section discusses some alternative algorithms and approaches we considered, but haven't explored. For proofs in other subjects, such as mathematics, software, or protocols, or other styles of proofs, different selection functions may be better. Additionally term recognition may be generally useful in developing tactics. So we mention some alternate approaches here.

4.1 Pattern Terms and General Match Function

Recognition functions can get verbose. Using pattern terms with a general match function yields more succinct functions, but much slower execution. For example using a general match function and a pattern the level 2 top abstraction of `"!x y. x < 3 /\ y > 0"` is `tmatch "!STAR1 STAR2. STAR3 /\ STAR4"`. The function `tmatch` checks each part of the pattern against the corresponding part of the term. This is used in the implementation for flexibility, but patterns are converted to functions for speed.

If the terms to be selected contain some particular function somewhere in them, a search-and-match function could be used. If `subtmatch` searches for a matching subterm, `subtmatch "lsim(STAR):num -> bool" t` searches `t` checking for a combination with `lsim` as the operator.

4.2 Other AI Techniques

The way we apply rule-version space search misses common subexpressions in positive examples. Common subexpression location with DAGs, as in compilers [1], is probably not useful since compilers look for *exact* matches. Discrimination networks [4] may be a means of finding common subexpressions. The common subexpressions could then be found with a search-and-match function.

The work of Feng and Muggleston [8] may be applicable. They are concerned with finding selectors which classify positive and negative examples of higher order terms. The statistical feature selection of Kira and Rendell [11] is another way of extracting concepts from large numbers of features. This is especially applicable since the set of potential features is the power set of term nodes.

4.3 Methods of Synthesizing Selection Predicates

The current method uses abstractions from the top down as features for building the selection function. Other kinds of features may lead to better selection functions.

Bottom-Up and General Abstraction Rather than starting at the top and allowing detail downward, details from the bottom of the term's parse tree upward might be used. A reasonable, informal definition of the level n abstraction from the bottom may be a parse tree of height n above any leaf node. Since constants and variables names are rarely good identifiers, all leaf nodes may be replaced with `STAR`'s. In contrast to top abstraction in Sect. 3.2 the level 1 abstractions from the bottom for the term `"!x y.x < 3 /\ y > 0"` are `"STAR1 < STAR2"` and `"STAR3 > STAR4"`. The level 2 bottom abstraction is `"STAR1 < STAR2 /\ STAR3 > STAR4"`.

Clearly one could search for and check abstractions found in the middle, too. For example, another pattern for the above term is `"STAR1 /\ STAR2"`. It is not clear how features could be chosen efficiently when terms are large. Since there are $O(2^n)$ nodes in a term of depth n, the total number of features grows as $O(2^{2^n})$.

Maximum Distance An entirely different approach is to define a metric of distance between terms in some multi-dimensional space. Denotation predicates would then specify hyperrectangles or separating hyperplanes. The synthesizing function would choose those predicates which maximize separation between the positive and negative examples.

5 Examples

5.1 Manufactured Examples

In order to illustrate the operation of `find_filter`, I present some small, contrived examples.

Simplest Suppose the assumption list at some point is

```
["!x. x /\ y ==> y /\ x";
 "!a. a /\ b"]
```

The following finds a filter for the first assumption:

```
find_filter (fst (top_goal())) [1];;
`let f = \(t:term).is_forall t & is_imp (body (rand t)) ? false;;`
: string
```

If one of the negative examples also has an implication, the match must be more exact.

```
find_filter ["!x. x /\ y ==> y /\ x";
             "!a. a /\ b";
             "!x. x ==> d"] [1];;
`let f = \(t:term).is_forall t & is_imp (body (rand t))
             & is_conj (rand (rator (body (rand t)))) ? false;;`
: string
```

A filter for two very different terms is a conjunction.

```
find_filter ["!x. x /\ y ==> y /\ x";
             "!x. a /\ b";
             "!x. x ==> d"] [2;3];;
`let f = \(t:term).is_forall t & is_conj (body (rand t))
          or is_forall t & is_imp (body (rand t))
             & vname (rand (rator (body (rand t))))=`x`
 ? false;;`: string
```

Clearly the duplicated test of `is_forall t` could be done once to shorten and speed the predicate.

5.2 Examples from Uinta

In this section we show `find_filter` used with one of biggest, most complex assumption list we could find in a real proof. In the proof of the general interpreter in Uinta [15], there is a point where a goal has 21 fairly complex assumptions. To save space, only a few of the assumptions are used and shown here. The assumptions shown are typical. These assumptions are *not* shown with enough type information to be reentered: that would make them even bigger. To get enough type information, print terms with `set_flag(`show_types`, true)`.

First Example We begin by synthesizing a selector for one of the most complex assumptions. The assumption is

```
"!t.
  (select
   gi2
   (substate
    gi2
    (substate gi1(s'(Temp_Abs(\t'. sync gi1(s' t')(e' t'))t))))
   (subenv gi2(subenv gi1(e'(Temp_Abs(\t'.
                                sync gi1(s' t')(e' t'))t)))) =
    k) /\
  sync
  gi2
  (substate gi1(s'(Temp_Abs(\t'. sync gi1(s' t')(e' t'))t)))
  (subenv gi1(e'(Temp_Abs(\t'. sync gi1(s' t')(e' t'))t))) ==>
  (subout gi2(subout gi1(p'(Temp_Abs(\t'.
```

```
                              sync gi1(s' t')(e' t'))t))) =
   output
   gi2
   k
   (substate
    gi2
    (substate gi1(s'(Temp_Abs(\t'. sync gi1(s' t')(e' t'))t))))
   (subenv gi2(subenv gi1(e'(Temp_Abs(\t'.
                              sync gi1(s' t')(e' t'))t)))))"
```

When we run `find_filter`, we get

```
\t.is_forall t & is_imp (body (rand t)) &
    is_conj (rand (rator (body (rand t))))
? false
```

The selection function may be informative by itself: it shows that at an abstract level, the structure of the assumption is `!STAR1. STAR2 /\ STAR3 => STAR4`.

Second Example In this example we use `find_filter` to find a selector for two structurally similar assumptions. The assumptions are

```
"(!s' e' p' k. INST_CORRECT gi2 s' e' p' k) ==>
 (!s' e' p' k. OUTPUT_CORRECT gi2 s' e' p' k) ==>
 (!s' e' p'.
   implementation gi2 s' e' p' ==>
   (!t.
     sync gi2(s' t)(e' t) ==>
     (?n. Next(\t'. sync gi2(s' t')(e' t'))(t,t + n))))"
```

```
"(!s' e' p' k. INST_CORRECT gi1 s' e' p' k) ==>
 (!s' e' p' k. OUTPUT_CORRECT gi1 s' e' p' k) ==>
 (!s' e' p'.
   implementation gi1 s' e' p' ==>
   (!t.
     sync gi1(s' t)(e' t) ==>
     (?n. Next(\t'. sync gi1(s' t')(e' t'))(t,t + n))))"
```

The selector is simply `\t.is_imp t`. No other assumptions are implications at the top.

Third Example The last example finds a filter for one of the simplest assumptions:

```
"implementation gi1 s' e' p'"
```

There is another similar assumption (shown below), so the filter must be very specific.

```
"implementation
 gi2
 (\x. substate gi1(s'(Temp_Abs(\t. sync gi1(s' t)(e' t))x)))
 (\x. subenv gi1(e'(Temp_Abs(\t. sync gi1(s' t)(e' t))x)))
 (\x. subout gi1(p'(Temp_Abs(\t. sync gi1(s' t)(e' t))x)))"

\t.cname (rator (rator (rator (rator t)))) = `implementation` &
    vname (rand (rator (rator (rator t)))) = `gi1`
    ? false
```

6 Conclusions

Until assumptions can be labeled, filtering the assumption list is the most robust way to write proof scripts. Having an aid to synthesize filter functions will encourage people to use filters instead of denoting assumptions by position. With a large user group, more experience, and a wider range of styles, more hueristics can be captured and the filter synthesizer improved. In addition, machine learning applied to terms may be the basis for other operations or proof functions.

Acknowledgements

We thank Tony R. Martinez for his suggestions about AI sources. We are indebted to the reviewers whose extensive comments pointed out vague and confusing parts of the paper.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley Publishing Co., 1986.
2. Jim Alves-Foss. message of 1 November 1991 to info-hol mailing list. (Available electronically at http://lal.cs.byu.edu/lal/hol-documentation.html)
3. Jaime G. Carbonell, Ryszard S. Michalski, and Tom M. Mitchell. (Eds.) *Machine Learning. An Artificial Intelligence Approach.* Tioga Publishing Company, Palo Alto, California, 94302, 1983, page 9.
4. Eugene Charniak, Christopher K. Riesbeck, and Drew V. McDermott. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, 1980, chapters 11 and 14, pp. 121–130 and 162–176.
5. Ching-Tsun Chou. "A Note on Interactive Theorem Proving with Theorem Continuation Functions," in *Higher Order Logic Theorem Proving and Its Applications (HOL '92)*, edited by Luc Claesen and Michael Gordon, Elsevier Science Publishers B.V., 1992, pp. 59–69.
6. Paul Curzon. *Re: Accessing Assumptions.* message of 25 May 1994 to hol2000 mailing list. (Available electronically at http://lal.cs.byu.edu/lal/hol-documentation.html)

7. M. J. C. Gordon and T. F. Melham. *Introduction to HOL. A theorem proving environment for higher order logic.* Cambridge University Press, 1993, Section 24.5, pp. 384–396.
8. Cao Feng and Stephen Muggleston. "Towards Inductive Generalisation in Higher Order Logic," in *Machine Learning: Proceedings of the Ninth International Workshop (ML92)* edited by Derek Sleeman and Peter Edwards, Morgan Kaufmann Publishers, 1992, pp. 154–162.
9. John Harrison. *Selecting Assumptions.* message of 31 October 1991 to info-hol mailing list. (Available electronically at http://lal.cs.byu.edu/lal/hol-documentation.html)
10. Sara Kalvala, Myla Archer, Karl Levitt. "Implementation and Use of Annotations in HOL," in *Higher Order Logic Theorem Proving and Its Applications (HOL '92)*, edited by Luc Claesen and Michael Gordon, Elsevier Science Publishers B.V., 1992, pp. 407–426.
11. Kenji Kira and Larry A. Rendell. "A Practical Approach to Feature Selection," in *Machine Learning: Proceedings of the Ninth International Workshop (ML92)* edited by Derek Sleeman and Peter Edwards, Morgan Kaufmann Publishers, 1992, pp. 249–256.
12. Miriam Leeser. *assumption numbering, user interfaces ....* message of 26 May 1994 to hol2000 mailing list, archived at http://lal.cs.byu.edu/lal/hol-documentation.html.
13. David Shepherd. *Re: Accessing Assumptions.* message of 26 May 1994 to hol2000 mailing list. (Available electronically at http://lal.cs.byu.edu/lal/hol-documentation.html)
14. Steven L. Tanimoto. *The Elements of Artificial Intelligence.* Computer Science Press, 1987.
15. Phillip J. Windley and Michael Coe. "A Correctness Model for Pipelined Microprocessors," in *Proceedings of the 1994 Conference on Theorem Provers in Circuit Design*, edited by Thomas Kropf and Ramayya Kumar, 1994.

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