What’s in a Theorem Name?

Citation for published version:

Digital Object Identifier (DOI):
10.1007/978-3-319-43144-4_28

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of ITP 2016: Interactive Theorem Proving 7th International Conference
What’s in a Theorem Name? (Rough Diamond)

David Aspinall\(^1\) and Cezary Kaliszyk\(^2\)

\(^1\) LFCS, School of Informatics, University of Edinburgh, UK.
\(^2\) Institut für Informatik, University of Innsbruck, Austria.

Abstract. ITPs use names for proved theorems. Good names are either widely known or descriptive, corresponding to a theorem’s statement. Good names should be consistent with conventions, and be easy to remember. But thinking of names like this for every intermediate result is a burden: some developers avoid this by using consecutive integers or random hashes instead. We ask: is it possible to relieve the naming burden and automatically suggest sensible theorem names? We present a method to do this. It works by learning associations between existing theorem names in a large library and the names of defined objects and term patterns occurring in their corresponding statements.

1 Introduction

How do we name theorems? Science has a tradition of historical reference, attaching names by attribution to their discoverer. HOL Light contains fine examples such as RIEMANN_MAPPING_THEOREM:

\[
\forall s. \text{open } s \land \text{simply_connected } s \iff s = \{\} \lor s = (\mathbb{R}^2) \lor \\
\exists f, g. f \text{ holomorphic_on } s \land g \text{ holomorphic_on } \text{ball}(0,1) \land \\
(\forall z. z \text{ IN } s \implies f z \text{ IN } \text{ball}(Cx(\&0), &1) \land g(f z) = z) \land \\
(\forall z. z \text{ IN } \text{ball}(Cx(\&0), &1) \implies g z \text{ IN } s \land f(g z) = z)
\]

Mere lemmas are seldom honoured with proper names. Papers and textbooks use localised index numbers instead (“see Lemma 5.7 on p.312”) which is succinct but unhelpful. For practical proof engineering, working with large developments and libraries, we are quickly swamped with intermediate lemmas. Despite search facilities in some ITPs, users still need to name statements, leading to the proliferation of less profound descriptive names such as:

\[
\text{ADD_ASSOC} : \forall m n p . m + (n + p) = (m + n) + p \\
\text{REAL_MIN_ASSOC} : \forall x y z . \text{min } x (\text{min } y z) = \text{min } (\text{min } x y) z \\
\text{SUC_GT_ZERO} : \forall x . \text{Suc } x > 0
\]

Descriptive names are convenient and mnemonic, based on the statement of a theorem, and following conventions due to the author, library or system. But inventing names requires thought, and remembering them exactly later can be tricky. Unsurprisingly, people complain about the burden of naming, even inventing schemes that generate automatic names like (in Flyspeck [4]):
Such (fuzzy) hashes of theorem statements or sequential numbers give no clue of content, and we are back to old-fashioned indices. Some readers may be happy, accepting that names should have “denotation but not connotation” (as Kripke recalled the position of Mill [6]). But like Kripke, we see value in connotation: after all, a name is the handle for a potentially much longer statement.

So we wonder: is it possible to relieve theorem naming burden by automatically naming theorems following established conventions? Given that we have large corpora of carefully named theorems, it is natural to try a learning approach.

2 Parts of names and theorems

To start, we examine the form of human-generated descriptive theorem names. These are compound with separators: $l_0 \ldots l_m$, where the $l_i$ are commonly used stem words (labels). Examples like REAL_MIN_ASSOC show a connection between names of constants (MIN), their types (REAL), and the structure of the statement (ASSOC). Using previous work on features for learning-assisted automated reasoning [5], we extract three characterizations of theorem statements:

- **Symbols**: constant and type names (including function names);
- **Subterms**: parts of the statement term where no logical operators appear;
- **Patterns**: subterms, with abstraction over names of defined objects.

Patterns allow us to model certain theorem shapes, such as commutativity, associativity, or distributivity, without the actual constants these properties talk about [2]. For examples like SUC_GT_ZERO, we see that where the name parts occur is important. So we also collect:

- **Positions**: for each feature $f$, a position $p(f)$, normalised so that $0 \leq p(f) \leq 1$, given by the position of $f$ in the print order of the statement.

The leftmost feature has position 0 and the rightmost 1; if only a single feature is found, it has position $\frac{1}{2}$. Names are treated correspondingly: for $l_0 \ldots l_m$, the stems are assigned equidistant positions $p(l_0) = 0, \ldots, p(l_m) = 1$.

3 Learning associations between names and statements

We investigate two schemes for associating theorem statements with their names:

- **Consistent**: builds an association between symbols (e.g., constant and type names) and parts of theorem names;
- **Abstract**: uses patterns to abstract from concrete symbols, building a matching between positions in statements and name parts.
We hypothesise that the first scheme might be the more successful when used within a specific development, (hopefully) consistently re-using the same sub-components of relevant names, whereas the second may do better across different developments (perhaps ultimately, even across different provers).

To try these schemes out, we implement a k-Nearest Neighbours (kNN) multi-label classifier. A proposed label is computed together with a weighted average of positions. The algorithm first finds a fixed number \( k \) of training examples (named theorem statements), which are most similar to a set of features being considered. The stems and positions from the training examples are used to estimate the relevance of stems and proposed positions for the currently evaluated statement. The \textit{nearness} of two statements \( s_1, s_2 \) is given by

\[
n(s_1, s_2) = \sqrt{\sum_{f \in T(s_1) \cap T(s_2)} w(f)^2}
\]

where \( w(f) \) is the IDF (inverse document frequency) weight of a feature \( f \). To efficiently find nearest neighbours, we index the training examples by features, so we can ignore examples that have no features in common with the currently considered statement. Given the set of \( k \) training examples nearest to the current statement, we evaluate the relevance of a label as follows:

\[
R(l) = \sum_{s_1 \in N \cap T(s_1)} \frac{n(s_1, s_2)}{|T(s_1)|}
\]

We propose positions for a stem using the weighted average of the positions in the recommendations; weights are the corresponding nearness values.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Frequency</th>
<th>Position</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>((V_0 + (V_1 + V_2)) = (V_0 + V_1) + V_2)</td>
<td>1</td>
<td>0.37</td>
<td>7.82</td>
</tr>
<tr>
<td>(((V_0 + V_1) + V_2))</td>
<td>1</td>
<td>0.75</td>
<td>7.13</td>
</tr>
<tr>
<td>((V_0 + V_1))</td>
<td>1</td>
<td>0.84</td>
<td>3.95</td>
</tr>
<tr>
<td>+</td>
<td>4</td>
<td>0.72</td>
<td>2.62</td>
</tr>
<tr>
<td>num</td>
<td>3</td>
<td>0.21</td>
<td>1.15</td>
</tr>
<tr>
<td>=</td>
<td>1</td>
<td>0.43</td>
<td>0.23</td>
</tr>
<tr>
<td>(\forall)</td>
<td>3</td>
<td>0.15</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Table 1. Selected features extracted from the statement of ADD_ASSOC.

As an example, we examine how the process works with the consistent naming scheme and the ADD_ASSOC statement shown previously. Our algorithm uses the statement in a fully-parenthesised form with types attached to binders:

\[
\forall\text{num}.(\forall\text{num}.((V_0 + (V_1 + V_2)) = (((V_0 + V_1) + V_2))))
\]

From this statement, features are extracted, computing their frequency, average position and then IDF across the statement. A total of 46 features are extracted;
Table 2. Nearest neighbours found for ADD_ASSOC.

Table 3. Stems and positions suggested by our algorithm, sorted by relevance measure.

Unsurprisingly, the consistent scheme performs less well in situations where new defined objects appear in a statement, it can only suggest stems it has seen before. The abstract scheme addresses this. For this, we first gather all symbol names in the training examples and order them by decreasing frequency. Next, for every training example we find the non-logical objects that appear, and replace their occurrences by object placeholders, with the constants numbered in the order of their global frequencies. For example, a name like DIV_EQ_0 becomes C0_EQ_C3, and the theorem statement is abstracted similarly.
For the statement of the theorem ADD_ASSOC the first three names predicted by the abstract naming scheme are: +_ASSOC, num_+, and num_+_ASSOC. The use of the stem ADD is only predicted with $k > 20$.

4 Preliminary evaluation

We perform a standard leave-one-out cross-validation to evaluate how good names predicted on a single dataset are. The predictor is trained on all the examples apart from the current one to evaluate, and is tested on the features of the current one.

For 2298 statements in the HOL Light core library, the results are presented in Table 4. We explored four different options for the algorithm:

– Upper: names are canonicalised to upper case.
– AbsN: we use the abstraction scheme described above for naming.
– AbsT: we use abstraction in statements before training.
– Stem: we use a stemming operation to break down names.

The first option is useful in the libraries of HOL Light and HOL4 where the capitalization is mostly uniform, but may not be desired in proof assistant libraries where this is not the case. For example min and Min are used with different semantics in Isabelle/HOL.

The last option allows us to model the naming convention in HOL Light where a statement is relative to a type, but the type name does not get repeated. For example, a theorem that relates the constants REAL_ABS and REAL_NEG can be given the name REAL_ABS_NEG rather than REAL_ABS_REAL_NEG.

Each row in Table 4 is a combination of options. Results are split in the columns: the number of statements for which the top prediction is the same as the human-given name (First Choice); the number where the human name is in place 2-10 (Later Choice); one of the ten names is correct modulo stem order (Same Stems); the number where the human-used stems are predicted but not combined correctly (All Stems); and the number where at least part of the prediction is correct (Stem Overlap). Altogether, the human-generated names are among the top ten predictions by some instance of the algorithm in over 50% of cases; 40% for the best performing version based on the abstract scheme. The abstract scheme beats the consistent mechanism even in the same library.

The final column shows the number of cases that fail completely. These include cases with familiar (historical) names such as:

– EXCLUDED_MIDDLE (proposed reasonable name: DISJ_THM)
– INFINITY_AX (proposed reasonable name: ONTO_ONE).

We would not expect to predict these names, unless they are already given in the training data. In other cases, failure might indicate inconsistency: the human names may not always be “correct”. We uncovered some cases of inconsistency such as where multiplication was occasionally called MULT rather than MUL, for example.
5 Conclusions

The initial results are encouraging and suggest foundations for name-recommender systems that might be built into ITP interfaces. Even if the perfect name is not proposed, suggestions may spark an idea for the user. We plan to go further and look at case studies such as renaming Flyspeck, or using naming maps as a bridge between different ITPs. Moreover, more advanced machine learning schemes could be used to distinguish the use of the same symbol for different operators.

Related work. This appears to be the first attempt to mine named theorems and produce a recommender system for naming new theorems in a meaningful way. There have been a number of non-meaningful proposals, e.g., Flyspeck’s random 8-character identifiers [4]; Mizar’s naming scheme of theory names and numbers (examples like WAYBEL34:67 [3]); the use of MD5 recursive statement hashes [8].

Identifier naming has been studied in software engineering. Lawrie et al [7] investigated name consistency in large developments. Deissenboeck and Pizka [1] build a recommender similar to our consistent scheme (but using different methods). They note that programming style guides say identifiers should be “self-describing” to aid comprehension. Indeed, program obfuscators, intended to hinder comprehension, randomize identifiers, producing names like those in Flyspeck.

Acknowledgments. This work has been supported by UK EPSRC (EP/J001058/1) and the Austrian Science Fund FWF (P26201).
References