Contents

1 Introduction **1**
- 1.1 Warum Term Indexing? . . . 1
- 1.2 In Isabelle/ML . . . 1
- 1.3 Vergleich mit Discrimination Net . . . 1
- 1.4 Implementation von QuickCheck . . . 1

2 Preliminaries **3**
- 2.1 Was genau ist Term Indexing? . . . 3
- 2.2 FOL, Isabelle ist eig HOL, Termrepräsentation . . . 3
- 2.3 Related work . . . 3

3 Term Indexing **5**
- 3.1 Discrimination Nets . . . 5
- 3.2 Path Indexing . . . 5
- 3.3 Implementation in Isabelle/ML . . . 7

4 QuickCheck **9**
- 4.1 Was gab es? . . . 9
- 4.2 Änderungen . . . 9
- 4.3 Probleme? . . . 9
- 4.4 Benchmarking (Falls das noch in Quickcheck landet, ansonsten eigenes Kapitel/in Evaluation?) . . . 9

5 Evaluation **11**
- 5.1 Testweise . . . 11
- 5.2 Vergleich von Queries, mit DN, Itemnets . . . 11
- 5.3 Vergleich von Insert, Delete, (Merge); mit DN, Itemnets, Termtabs . . . 11
- 5.4 Wann PI hernehmen statt dem Rest? . . . 11
- 5.5 Shortcomings . . . 11
- 5.6 Future Work . . . 11

6 Conclusion **13**

1 Introduction

1.1 Warum Term Indexing?

1.2 In Isabelle/ML

1.3 Vergleich mit Discrimination Net

1.4 Implementation von QuickCheck

2 Preliminaries

2.1 Was genau ist Term Indexing?

2.2 FOL, Isabelle ist eig HOL, Termrepräsentation

2.3 Related work

3 Term Indexing

A term index is used to store a collection of terms for efficient querying. The queries commonly include retrieval of the generalisations or unifiables of a term from the index.

Move to general section before DN and PI

In the following sections we give an overview of discrimination nets and path indexing, we also take a closer look at some details of their implementation in Isabelle/ML.

3.1 Discrimination Nets

3.2 Path Indexing

Instead of storing a term as a tree of functions and their arguments we can specify the structure and symbols of a tree by combining every symbol of a term with its position, which we call its path. This path is sequence of $(symbol, index)$ pairs where the index describes the argument which we traverse next.[1] The term $f(x, g(a, b))$ can thus be represented by a set of paths and their associated symbol. The paths always start at the root and end with the index at which the symbol is located, e.g. $< f \cdot 2 \cdot g \cdot 1 >$ is the path of the symbol a. We use $<>$ to enclose a path and $\cdot$ to express the sequence of $(symbol, index)$ pairs. We disregard the identity of variables as they are irrelevant to the queries. $Symbol_t(p)$ is used to refer to the symbol associated with path p.

Notation chapter? Don't use at all, copy definition or have definition only in notation chapter?

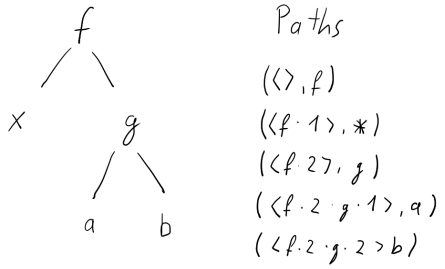

Figure 3.1: A term and its paths together with the symbols associated with them

Write as mapping: $<> \longrightarrow f$ etc.

This set of paths allows us to define terms not explicitly by their structure and symbols but rather by imposing constraints on them. A term must fulfill the constraint enforced by every path and its associated symbol. This is relevant to the structure of the path index but also to the queries where we drop certain constraints to retrieve multiple terms.

We store multiple terms in one index structure by mapping each $(path, symbol)$ pair to a set of terms that fulfill this constraint. We call these path sets. Storing the path sets such that they can be quickly looked up by the $(path, symbol)$ pair can be achieved

[1] This is in contrast to coordinate indexing which only uses a sequence of indices.

in multiple ways. We decided to use a trie-based approach as many of the paths share prefixes.

The nodes of the trie contain a mapping of symbols to path sets. The the edges are labelled with $(symbol, index)$ pairs which are the elements of the paths. When we insert a path p of a term t we start at the root and traverse the trie according to the elements of p. Once we reach the end of p we insert a mapping $Symbol_t(p) \longrightarrow t$ into the path set at that node. To insert a term we simply insert all the paths that describe this term. This requires the insertion of many similar paths which profits from the prefix sharing.

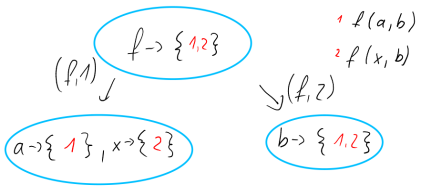

Figure 3.2: A term index

We are interested in retrieving the instances, generalisations and unifiables of a term stored in the index. Additionally to that we define a lookup to retrieve copies of the term. This can be used to check if a term is already contained but may also be of interest as different variables are not distinguished. The queries are based on intersections and unions of the different path sets to enforce constraints on the terms. To answer the simplest query, the lookup, we first compute the set of $(path, symbol)$ pairs describing the term. Next we retrieve the path sets corresponding to them in the index. The intersection of these path sets returns all terms containing symbols at identical paths as the query term. Under the assumption of consistent typing we retrieve only the original or no term as $f(x, y)$ can not exist simultaneously to $f(x)$.

To retrieve the unifiables of a term from the index we can use some observations regarding the unification problem.

1. A variable is unifiable with any other term

2. Constants are unifiable with themselves and variables

3. A function $f(x_1, ..., x_n)$ is unifiable with term t if t is a variable or a function $g(y_1, ..., y_n)$ where $f = g$ and for all i x_i is unifiable with y_i. Again, differing number of arguments are impossible as the type of f and g would clash otherwise.

Using this we can define an algorithm recursing on the structure of the query term while intersecting and unifying the different path sets of the index. The different query types are quite similar, with the lookup being the most restrictive and the unifiables the least restrictive.

$PathTerms(p)$ refers to the path set stored at the path p. $AllTerms$ is the collection of all terms stored in the index.

Arguments \ Query	Lookup	Instances	Generalizations	Unifiables
p x	$\text{PathTerms}(p \cdot *)$	AllTerms	$\text{PathTerms}(p \cdot *)$	AllTerms
p a	$\text{PathTerms}(p \cdot a)$	$\text{PathTerms}(p \cdot a)$	$\text{PathTerms}(p \cdot *) \cup \text{PathTerms}(p \cdot a)$	$\text{PathTerms}(p \cdot *) \cup \text{PathTerms}(p \cdot a)$
p $f(t_1, \ldots, t_n)$	$\bigcap_n \text{Lookup}(p \cdot f \cdot n, t_n)$	$\bigcap_n \text{Instances}(p \cdot f \cdot n, t_n)$	$\text{PathTerms}(p \cdot *) \cup \bigcap_n \text{Generalisations}(p \cdot f \cdot n, t_n)$	$\text{PathTerms}(p \cdot *) \cup \bigcap_n \text{Unifiables}(p \cdot f \cdot n, t_n)$

Figure 3.3: The different queries and their definition

3.3 Implementation in Isabelle/ML

The discrimination net implementation already present in Isabelle is used to store arbitrary values indexed by terms. This allows us to store certified terms or context together with terms. The lookup operation mentioned earlier is also more useful in this context.

Unfortunately most literature on path indexing only covers the storage of terms. The queries of path indexing rely on the intersection and union of path sets. These in turn rely on the fast comparison of the stored values. To solve this potential problem we investigated multiple solutions.

2 Quellen sicher, noch welche?

The first approach requires a comparison function for the values[2]. Using the index becomes more difficult by doing so. A user has to implement a comparison function for values and additionally has to consider the potential performance impact. This can be partially mitigated by using *pointerEq* although it can only be used as a shortcut for identical values, the comparison must still be called for differing values.

The second approach would be the storage of $(term, value)$ pairs. By doing so we can implement all the operations according to the literature and simply discard the term before returning the results. This simplifies implementation and retains acceptable performance as the comparison of differing values will likely only need to compare the first few symbols. It will also increase the memory consumption as a copy of every term is stored solely for the set operations. (Additionally there is no immutable pointer implementation in Isabelle/ML. Instead, copies of identical values are shared by the runtime.)

PolyML erwähnen?

This approach can be further optimised by replacing the $(term, value)$ pairs by $(identifier, value)$ pairs and mapping each term to an identifier. By using integers as identifier we further speed up the comparison and can use ordered lists, provided by the SML standard library, for the path sets to implement the set operations more efficiently. We are also less reliant on the pointer equality provided by Poly/ML and runtime details like the merging of identical immutable values. This is quite important as we do not have any guarantee when the last heap compression occurred and manual invocation by using the "shareCommonData" introduces significant overhead to insertion. Additionally, reliance on low-level functions like "shareCommonData" and "pointereq" should be avoided as there are may be significant changes across runtime versions.

Überhaupt nicht erwähnen weil Data Sharing gut genug funktioniert?

[2]Either as an argument to every function or by implementing path indexing as a functor on a value module

We can further speed up the set operations by building a tree of the intersections and unions and only evaluating it at the end. This likely utilizes the cache better because the previously calculated list is not evicted from the cache by the trie traversal. Furthermore, this presumably enables further compiler optimizations as the intermediate results are only short-lived and functions can be inlined.

Data Sharing in Poly/ML. NoConstraint exc. No generic hash. Saving "Copy" of values because pointers/ref are always mutable and bad for GC etc.

4 QuickCheck

4.1 Was gab es?

4.2 Änderungen

4.3 Probleme?

4.4 Benchmarking (Falls das noch in Quickcheck landet, ansonsten eigenes Kapitel/in Evaluation?)

5 Evaluation

5.1 Testweise

5.2 Vergleich von Queries, mit DN, Itemnets

5.3 Vergleich von Insert, Delete, (Merge); mit DN, Itemnets, Termtabs

5.4 Wann PI hernehmen statt dem Rest?

5.5 Shortcomings

5.6 Future Work

6 Conclusion