Contents

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.

Definition 3.1. The path is a sequence of $(symbol, index)$ pairs where the index describes the index of the next argument to traverse.[1]

For example, the term $f(x, g(a, b))$ can be represented by a set of paths and their associated symbol as can be seen in fig. 3.1. The paths always start at the root and end with the index at which the symbol is located, e.g. $< (f, 2), (g, 1) >$ is the path of the symbol a. We represent a path by enclosing a sequence of $(symbol, index)$ pairs with $<>$.

As of now: No Notation chapter because there is almost none

We disregard the identity of variables as they are mostly irrelevant to the queries and simplify the queries. Consequently, the terms $f(x, y)$ and $f(x, x)$ are both saved as $f(*, *)$. This leads to an overapproximation in some cases, for example $f(c, d)$ would be identified as an instance of $f(x, x)$.

Definition 3.2. $Symbol_t(p)$ is used to refer to the symbol associated with path p in the term t.

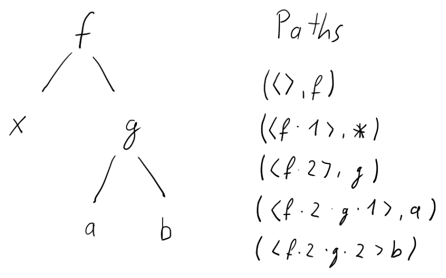


Figure 3.1: A term and its paths together with the symbols associated with them

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

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

A $(path, symbol)$ pair can be interpreted as a constraint on a term where the path defines the position of the symbol in the term. For example, $(<(f,1)>, c)$ is only fulfilled by terms of the form $f(c, ...)$[2]. A term gives raise to a set of $(path, symbol)$ pairs, which, when interpreted as constraints, uniquely identify this term up to loss of variable identitification.

These constraints allow us to define terms not explicitly by their structure and symbols but rather by imposing constraints on them. This concept is fundamental to path indexing which stores only the constraints. Queries are resolved by combining the constraints to retrieve a set of terms.

Better phrasing

Definition 3.3. A path index is a function $f : Path \times Symbol \longrightarrow 2^{Term}$, that is, each constraint is mapped to a set of terms, which fulfill these constraints and are stored in the path index. We call this set of terms a path set.

Drop "path set" completely?

Storing the path sets such that they can be quickly looked up by a $(path, symbol)$ pair can be achieved 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 function $g : Symbol \longrightarrow 2^{Term}$. The edges are labelled with $(symbol, index)$ pairs, which correspond to the elements of a path. When we insert a path p of a term t we start at the root and traverse the trie according to p. Once we reach the end of p we extend g of the current node by $Symbol_t(p) \longrightarrow \{t\}$. 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.

If already existant: Expand set, else extend definition of g to $Symbol_t(p)$

Figure 3.2 shows a path index stored as a trie. The root contains a mapping from the symbol f to both terms as they both share this constraint. In the first argument, reached by the edge $(f,1)$, the symbol a is mapped only to the first term whereas $*$ is mapped to the second term. In the second argument, the terms share the constraint.

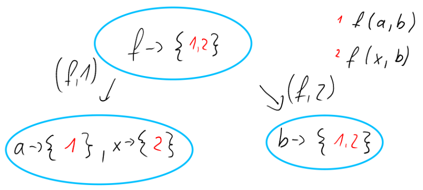


Figure 3.2: A path index

We are interested in retrieving the instances, generalisations and unifiables of a term stored in the index. In addition, 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 procede as follows:

1. Compute the set of $(path, symbol)$ pairs describing the term.
2. Retrieve the path sets corresponding to them in the index.
3. Intersect the path sets to retrieve all terms containing the same symbols at identical paths as the query term.

[2]The paths do not explicitly encode any information about the number of arguments.

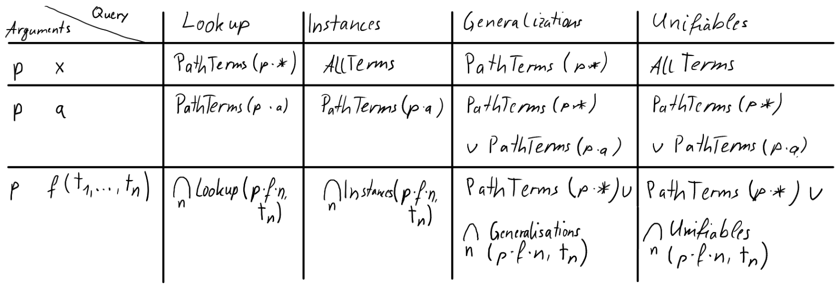
Under the assumption of consistent typing, we retrieve only terms of identical structure as $f(x, y)$ can not exist simultaneously to $f(x)$. Due to the loss of variable identity we may retrieve additional terms.

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 and only if $t = x$ or $t = f(y_1, ..., y_n)$ where for all i x_i is unifiable with y_i. Again, a differing number of arguments are impossible as their types 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.



Query / Arguments		Lookup	Instances	Generalizations	Unifiables
p	x	PathTerms (p·∗)	AllTerms	PathTerms (p∗)	All Terms
p	a	PathTerms (p ·a)	PathTerms (p·a)	PathTerms (p∗) ∪ PathTerms (p·a)	PathTerms (p∗) ∪ PathTerms (p·a)
p	$f(t_1, \dots, t_n)$	$\bigcap_n$ Lookup (p·f·n, t_n)	$\bigcap_n$ Instances (p·f·n, t_n)	PathTerms (p·∗) ∪ $\bigcap_n$ Generalisations (p·f·n, t_n)	PathTerms (p·∗) ∪ $\bigcap_n$ Unifiables (p·f·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[3]. Using the index becomes more difficult by doing so. A user has to implement a comparison function for

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

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?

Überhaupt nicht erwähnen weil Data Sharing gut genug funktioniert?

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.

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