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 Path 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 this section we give an overview of path indexing and take a closer look at some details of its implementation in Isabelle/ML.

3.1 Theory

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?

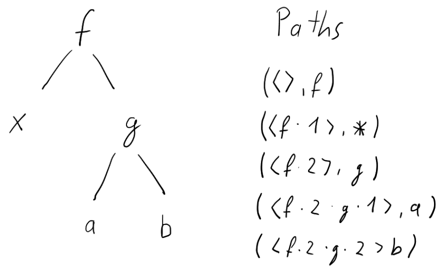


Figure 3.1: A term and its paths together with the symbols associated with them

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

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

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.

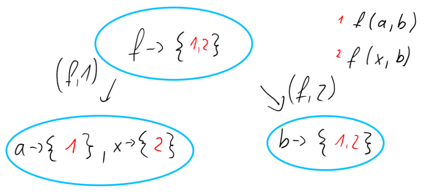


Figure 3.2: A term index

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.

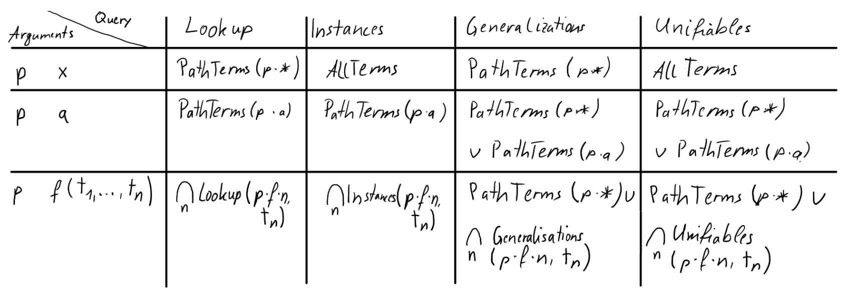
Using this we can define an algorithm recursing on the structure of the query term while intersecting and unifying the different path sets of the index. The different query types are quite similar, with the lookup being the most restrictive and the unifiables the least restrictive.

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

rename to PathSet

3.2 Implementation in Isabelle/ML

The efficient implementation of path indexing relies on a number of optimizations. The most common and performance critical operations are the queries which in turn rely on the



Arguments / Query	Lookup	Instances	Generalizations	Unifiables
$p \quad x$	PathTerms $(p \cdot *)$	AllTerms	PathTerms $(p *)$	All Terms
$p \quad a$	PathTerms $(p \cdot a)$	PathTerms $(p \cdot a)$	PathTerms $(p *)$ $\cup$ PathTerms $(p \cdot a)$	PathTerms $(p *)$ $\cup$ PathTerms $(p \cdot a)$
$p \quad f(t_1, \dots, t_n)$	$\bigcap_n$ Lookup$(p \cdot f \cdot n, t_n)$	$\bigcap_n$ Instances$(p \cdot f \cdot n, t_n)$	PathTerms $(p \cdot *)$ $\cup$ $\bigcap_n$ Generalisations $(p \cdot f \cdot n, t_n)$	PathTerms $(p \cdot *)$ $\cup$ $\bigcap_n$ Unifiables $(p \cdot f \cdot n, t_n)$

Figure 3.3: The different queries and their definition

union and intersection of the path lists. Unfortunately the interface of the already existing discrimination net allows the storage of arbitrary values addressed by terms. While this is unproblematic for discrimination nets it complicates path indexing.

First of all, the values stored in the path lists must be comparable to define set operations on them. Secondly, the comparison must also be extremely fast. While Poly/ML, the compiler on which Isabelle/ML is built, provides a pointer equality function, which is defined for any type, it does not guarantee equality for identical immutable values. Storing the values in mutable ref cells is also not an option as the Poly/ML runtime has significant overhead associated with them, both in memory consumption and garbage collection time.

Therefore we were presented with two options. Either we require the values to be comparable or we store an identifier together with every value at insertion where we still have the context to determine which paths belong to a value. The first approach prevents the path index from being used as a drop-in replacement for discrimination nets and complicates the usage, especially regarding an efficient comparison on complex data types. The second approach requires some additional memory to store the identifiers but offers multiple benefits. The user need not specify a comparison function or worry about its performance. We can rely on integer equality which is naturally extremely fast and need not use the pointer equality provided by Poly/ML which is reliant on 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.

Another advantage of integer identifiers is that they are ordered. This allows us to use the OrdList module of the SML standard library for path lists and further speeds up the set operations.

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 takes better advantage of the cache 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.

Generic values. Data Sharing in Poly/ML. Intersect at end for cache. Efficient intersect with OrdList. pointereq. Saving values separately. NoConstraint exc. No generic hash

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