

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 | First-Order Logic | 3 |
| 2.1.1 | Unification | 3 |
| 2.2 | Isabelle | 3 |
| 2.2.1 | Term Representation in Isabelle | 3 |
| 2.3 | Term Indexing | 4 |
| 2.4 | Related work | 4 |
| 3 | Term Indexing | 5 |
| 3.1 | Discrimination Nets | 5 |
| 3.2 | Path Indexing | 7 |
| 3.3 | Implementation in Isabelle/ML | 9 |
| 4 | QuickCheck | 13 |
| 4.1 | Was gab es? | 13 |
| 4.2 | Änderungen | 13 |
| 4.3 | Probleme? | 13 |
| 4.4 | Benchmarking (Falls das noch in Quickcheck landet, ansonsten eigenes Kapitel/in Evaluation?) | 13 |
| 5 | Evaluation | 15 |
| 5.1 | Testweise | 15 |
| 5.2 | Vergleich von Queries, mit DN, Itemnets | 15 |
| 5.3 | Vergleich von Insert, Delete, (Merge); mit DN, Itemnets, Termtabs | 15 |
| 5.4 | Wann PI hernehmen statt dem Rest? | 15 |
| 5.5 | Shortcomings | 15 |
| 5.6 | Future Work | 15 |
| 6 | Conclusion | 17 |
| | Bibliography | 19 |

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 First-Order Logic

First-order logic is a formal language used to, amongst others, formalise reasoning, including artificial intelligence, logic programming and automated deduction systems. In this thesis we are only interested in terms. Therefore, we disregard formulas, relations and quantifiers.

A term in first-order logic is either a variable, a constant or a function applied to some arguments. All symbols have arity 0. A function f has a fixed arity $a(f) \geq 1$ and is applied to an $a(f)$ -tuple of terms. We notate terms with lower-case letters with constants using letters from a to e , functions from f to j and variables from u to z .

2.1.1 Unification



2.2 Isabelle

Isabelle is an interactive theorem prover supporting many formal theories, including first-order and higher-order logics. The basic datatype used for terms are a variant of typed λ -calculus. Therefore, encoding first-order logic terms in the applicative λ -calculus becomes necessary.¹

The prover in Isabelle repeatedly uses unification to advance the proof. It starts with a goal, whose validity must be shown, and a list of inference rules, given as horn clauses, i.e. consisting of a conjunction of premises and a conclusion. The prover now continually attempts to unify the goal with the conclusion of one of the inference rules. If successful, it replaces the goal with the premises, each of which represent a subgoal.

Upon showing the validity of each subgoal the prover is finished. If no further unification is possible the algorithm backtracks and attempts to continue with a different inference rule. Depending on the number and form of inference rules available the prover will at some point either fail to satisfy the goal or continue searching forever. Tuning the prover to provide a proof while also avoiding both failure states is one of the major challenges in automated theorem proving.[\[1\]](#)

2.2.1 Term Representation in Isabelle

A λ -term in Isabelle is defined recursively in an applicate style. The different constructors are:

1. Free: Symbols that can not be instantiated.

¹We disregard types as we assume the user of the index to only store consistent and type-correct terms

2. Const: Symbols that are instantiated once and therefore are, within a given context, constant.
3. Var: Variables that have not been instantiated.
4. Abs and Bound: Abstractions, or λ -expression, instantiate the symbols bound to them with the argument they are applied to. A bound symbol stores a reference to the abstraction it is bound to.
5. App: Applications apply their first argument to the second argument. We use $\$$ as an infix operator.

Draw as tree, too many braces

For example, the λ -term $(\lambda x.f(x))c$ can be encoded as $App(Abs(x, App(Free(f), Bound(x))), Const(c))$.

Using this datatype, we can embed first-order terms in λ -terms. A constant c is equivalent to $Const(c)$. Likewise, a variable x is equivalent to $Var(x)$. Functions are written in the applicative form, e.g. $f(x, c)$ is encoded as $Const(f) \$ Var(x) \$ Const(c)$.

Obviously, this embedding is not invertible. Nevertheless, it is desirable for term indices to handle higher-order logic terms gracefully. We therefore

Motivation, move elsewhere?

2.3 Term Indexing

In automated theorem proving there are, generally, many inference rules available while only few are applicable to the current subgoals. Therefore, we are interested in efficient storage of inference rules, both in regards to memory consumption and fast retrieval of rules with unifiable premises. One category of data structures fit for this purpose are the term indices.

A term index efficiently stores terms while also providing efficient operations for the retrieval of instances, generalisations and unifiabiles of a query term from the term index. Depending on the exact data structure, we can also implement other operations efficiently if desired, e.g. checking if a term is already stored, only inserting a term if no more general term is stored, efficiently merging two indices etc.

As it is difficult to predict the use cases of the term indices we only implement the basic query operations.

already implemented for the discrimination nets

2.4 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.

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.

Move to general section before DN and PI

3.1 Discrimination Nets

A discrimination-net index, also known as discrimination-tree index, is a trie which stores terms at its leaves and symbols at its internal nodes. To determine the leaf at which a term is stored we use the preorder traversal of the term which we obtain by simply reading the written term from left to right.

Discrimination net = trie aber nur noch trie wird heutzutage verwendet?

Definition 3.1. $Preorder(t)$ is the sequence of symbols obtained by a preorder traversal of the term t . For terminals it is the symbol itself. The preorder traversal of function $f(x_1, \dots, x_n)$ is $< f, Preorder(x_1), \dots, Preorder(x_n)$.

For example, the preorder traversal of $t = f(c, g(x, y))$ is $< f, c, g, *, * >$ ¹.

We store the mapping $Preorder(t) \rightarrow t$ in a trie. Due to prefix sharing among the preorders of the different terms this reduces the memory consumption. The leafs contain the terms while the internal nodes only store the symbol by which they are addressed. A trie storing multiple terms can be seen in 3.1.

No term is stored in an internal node as, under the assumption of type consistency, it is impossible for $Preorder(t)$ to be a prefix of $Preorder(u)$ if $t \neq u$. If $Preorder(t)$ is a prefix of $Preorder(u)$, u is a combination of t and some other terms. All the functions and their arguments in t are also present in u . As a consequence, every function present in t and u is applied to all their arguments in u . Thus, u must not consist of any additional terms and therefore be identical to t .

Root $\neq f$, besser auf Beispiele ausgelegt: $f(c, x)$ enthalten, netskip kommt vor etc.

The queries are implemented as a recursive algorithm on the nodes of the trie and the term. Starting at the root, we traverse the tree by selecting the child node corresponding to the current symbol of the term. We recursively continue at the child node while removing the current symbol from the term. The symbol we remove may have arguments. Therefore, we store the arguments in a list and continue with the first argument.²

Definition 3.2. $slp(N, c)$ is the symbol lookup operation. It returns the child node of N containing the symbol c .

Retrieving the variants of a term is fairly straightforward. Starting at the root, we traverse the trie according to the term using slp . By doing so, we ensure that we only

¹We ignore variable identity as it introduces significant complexity

²We do not use $Preorder(t)$ as the preorder traversal does not retain the arity of functions

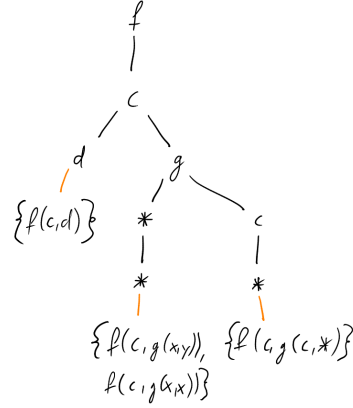


Figure 3.1: Multiple terms stored in a trie and indexed by the sequence obtained from their preorder traversal

retrieve terms that contain the same symbols (disregarding variable identity) in the same order, i.e. the variants of the term.

For example, we retrieve the variants of the term $t = f(c, x)$ in the discrimination net 3.1 in the following manner: $(root, < f(c, x) \rightarrow (f, < c, x >) \rightarrow (c, < x >) \rightarrow (*, < >)$ At this point we have reached a leaf containing the term $f(c, x)$, i.e. the only stored variant of the term.

The other queries are more intricate as they may now replace variables by arbitrary terms or symbols by arbitrary terms, with unification combining both. For every constant symbol in the term we form the union of both the query on the node $slp(N, c)$ as well as $slp(N, *)$. This ensures that indexed terms containing variables are also retrieved.

Likewise, a variable in the query term must also be handled differently. As the variable may be replaced by arbitrary terms we must skip a number of nodes depending on the arity of the symbol.

Definition 3.3. $skip(N)$ returns the set of nodes obtained by skipping a single term starting at N . For all symbols s for which $slp(N, s)$ is defined we collect all nodes $arity(n)$ levels below N . That is, for a constant c with $arity(c) = 0$ we return $slp(N, c)$ (which is a direct child of N). For a unary function f we return all the nodes $slp(sl p(N, f), x)$ defined for some x .

Using this, we can retrieve all the nodes reached by replacing the variable in the query term with some term. The union of the terms returned by the query on each node represents the result. An overview of all the queries is given in 3.2. Note that *variants* is the simplest and most restrictive query, *unifiables* is the most complex and least restrictive with *instances* and *generalisations* being a combination of both.

Definition 3.4. $Symbol_t(p)$ refers to the symbol associated with the preorder traversal p of the term t upto the symbol.

For example, the symbols of the term $t = f(c, g(x, y))$ are described by the following mapping:

1. $Symbol_t(< >) = f$

Symbol_t seems completely superfluous, why is it introduced in another paper?

| | $Q(\mathcal{V}, \langle \rangle)$ | $Q(\mathcal{M}, \langle x, t_1, \dots, t_m \rangle)$ | $Q(\mathcal{M} \wedge q, \langle t_1, \dots, t_m \rangle)$ | $Q(\mathcal{M}, \langle f(x_1, \dots, x_n), t_1, \dots, t_m \rangle)$ |
|------------------------------|-----------------------------------|---|--|---|
| $Q = \text{variants}$ | $\text{terms}(\mathcal{V})$ | $\bar{Q}(\mathcal{A}p(\mathcal{M}, *), \langle t_1, \dots, t_m \rangle)$ | $\bar{Q}(\mathcal{A}p(\mathcal{M}, a), \langle t_1, \dots, t_m \rangle)$ | $\bar{Q}(\mathcal{A}p(\mathcal{M}, f), \langle x_1, \dots, x_n, t_1, \dots, t_m \rangle)$ |
| $Q = \text{instances}$ | $\text{terms}(\mathcal{V})$ | $\bigcup_{\mathcal{M} \in \text{shd}(\mathcal{V})} Q(\mathcal{M}, \langle t_1, \dots, t_m \rangle)$ | $\bar{Q}(\mathcal{A}p(\mathcal{M}, a), \langle t_1, \dots, t_m \rangle)$ | $\bar{Q}(\mathcal{A}p(\mathcal{M}, f), \langle x_1, \dots, x_n, t_1, \dots, t_m \rangle)$ |
| $Q = \text{generalizations}$ | $\text{terms}(\mathcal{V})$ | $\bar{Q}(\mathcal{A}p(\mathcal{M}, *), \langle t_1, \dots, t_m \rangle)$ | $\bar{Q}(\mathcal{A}p(\mathcal{M}, *), \langle t_1, \dots, t_m \rangle) \vee \bar{Q}(\mathcal{A}p(\mathcal{M}, a), \langle t_1, \dots, t_m \rangle)$ | $\bar{Q}(\mathcal{A}p(\mathcal{M}, *), \langle t_1, \dots, t_m \rangle) \vee \bar{Q}(\mathcal{A}p(\mathcal{M}, f), \langle x_1, \dots, x_n, t_1, \dots, t_m \rangle)$ |
| $Q = \text{unifiable}$ | $\text{terms}(\mathcal{V})$ | $\bigcup_{\mathcal{M} \in \text{shd}(\mathcal{V})} Q(\mathcal{M}, \langle t_1, \dots, t_m \rangle)$ | $\bar{Q}(\mathcal{A}p(\mathcal{M}, *), \langle t_1, \dots, t_m \rangle) \vee \bar{Q}(\mathcal{A}p(\mathcal{M}, a), \langle t_1, \dots, t_m \rangle)$ | $\bar{Q}(\mathcal{A}p(\mathcal{M}, *), \langle t_1, \dots, t_m \rangle) \vee \bar{Q}(\mathcal{A}p(\mathcal{M}, f), \langle x_1, \dots, x_n, t_1, \dots, t_m \rangle)$ |

Figure 3.2: The definition of the different queries

2. $\text{Symbol}_t(< f >) = c$
3. $\text{Symbol}_t(< f, c >) = g$
4. $\text{Symbol}_t(< f, c, g >) = *$
5. $\text{Symbol}_t(< f, c, g, * >) = *$

This mapping can be efficiently stored in a discrimination net or trie by storing one symbol of the preorder traversal in each internal node and storing

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 3.5. The path is a sequence of $(\text{symbol}, \text{index})$ pairs where the index describes the index of the next argument to traverse.³

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. 3.3. 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 $(\text{symbol}, \text{index})$ pairs with $<>$.

We disregard the identity of variables as they are mostly irrelevant to the queries and simplify both the index and 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 3.6. $\text{Symbol}_t(p)$ refers to the symbol associated with path p in the term t .

A $(\text{path}, \text{symbol})$ 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 $f(c, \dots)$ ⁴. A term gives raise to a set of $(\text{path}, \text{symbol})$ pairs, which, when interpreted as constraints, uniquely identify this term up to loss of variable identification.

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

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

³This is in contrast to coordinate indexing which only uses a sequence of indices.

⁴The paths do not explicitly encode any information about the number of arguments.

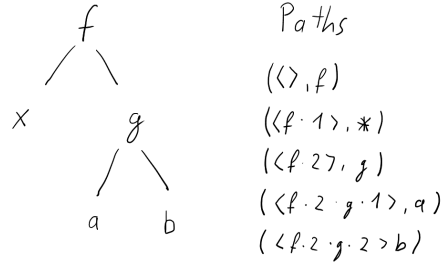


Figure 3.3: A term and its paths together with the symbols associated with them

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.

Definition 3.7. A path index is a function $f : Path \times Symbol \rightarrow 2^{Term}$, 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.

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 $g : Symbol \rightarrow 2^{Term}$. 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 $Symbol_t(p) \rightarrow \{t\}$. 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.

Figure 3.4 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.

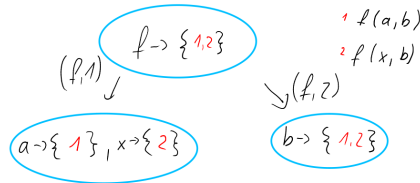


Figure 3.4: A path index

We are interested in retrieving the instances, generalisations and unifiers 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 proceed 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.

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 $f(x_1, \dots, x_n)$ is unifiable with term t if and only if $t = x$ or $t = f(y_1, \dots, y_n)$ 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.

| Arguments \ Query | Lookup | Instances | Generalization | Unifiables |
|------------------------------|---|--|---|--|
| $p \quad x$ | $\text{PathTerms}(p \cdot *)$ | AllTerms | $\text{PathTerms}(p \cdot *)$ | AllTerms |
| $p \quad a$ | $\text{PathTerms}(p \cdot a)$ | $\text{PathTerms}(p \cdot a)$ | $\text{PathTerms}(p \cdot *) \vee \text{PathTerms}(p \cdot a)$ | $\text{PathTerms}(p \cdot *) \vee \text{PathTerms}(p \cdot a)$ |
| $p \quad f(t_1, \dots, t_n)$ | $\bigcap_n \text{Lookup}(p \cdot f \cdot n, t_n)$ | $\bigcap_n \text{Instances}(p \cdot f \cdot n, t_n)$ | $\text{PathTerms}(p \cdot *) \vee \bigcap_n \text{Generalisations}(p \cdot f \cdot n, t_n)$ | $\text{PathTerms}(p \cdot *) \vee \bigcap_n \text{Unifiables}(p \cdot f \cdot n, t_n)$ |

Figure 3.5: 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 sets of values indexed by terms. This allows us to, for example, store certified terms or associate symbols with the terms that contain them. The lookup operation mentioned earlier now retrieves a terms associated set of values instead of only retrieving variants of the query term. Therefore, it becomes more useful in this context too.

Unfortunately most literature on path indexing only covers the storage of terms. The queries of path indexing rely on the intersection and union of the path sets. These in turn rely on the fast comparison of the stored values. For example, storing hash tables in the path index would be extremely slow as they cannot be compared directly⁵ and comparing the contents requires the collection of all entries. To solve this potential problem we investigated multiple solutions.

2 Quellen sicher, noch welche?

Furthermore, the insertion of an identical $(term, value)$ pair raises an exception. We require a comparison function during insertion to determine this.⁶ As the index only stores the terms in the path sets we have to store $(term, value)$ pairs in the path sets. Assume we store only the values in the path sets. We cannot determine whether, for example, the path index containing $(f(c), 1)$ and $(g(d), 1)$ has also stored the $(f(d), 1)$ as the value 1 is present in all path sets associated with $f(d)$, namely $(\langle \rangle, f)$ and $(\langle f, 1 \rangle, d)$.

The first approach requires a comparison function for the values.⁷ 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 since there is no perfect sharing.

The second approach is 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 terms 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.)

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 reduce the comparison to an integer comparison. Additionally, we 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

⁵The structure of the hashtable depends on the insertion order

⁶The comparison function need not be reflexive, for example the constant $(\lambda x. \text{false})$ is valid.

⁷Either as an argument to every function or by implementing path indexing as a functor on a value module

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

Bibliography

- [1] L. C. Paulson. “Isabelle: The Next 700 Theorem Provers”. In: (), p. 24.