Abstract

An important development for interactive theorem provers is the introduction of increasingly powerful proof automation. This automation necessitates efficient data structures to index and query sets of terms. The so-called term indices provide queries for retrieving variants, instances, generalisations and unifiables of a given term. Two indexing techniques, path indexing and discrimination tree indexing are reviewed. We implement path indexing for first-order terms in Isabelle/ML and adapt it to the more general term indexing interface defined in Isabelle/ML. We further define a unified interface for the previously implemented discrimination tree index and the path index, thereby clearing the way for additional term indices. Lastly, we evaluate the performance of path indexing relative to discrimination tree indexing.

Contents

1 Introduction

1.1 Motivation

Modern automated theorem provers can efficiently work on thousands of terms. One key ingredient for this is the effective use of efficient term indexing techniques. Each such technique for term indexing offers a different set of advantages and drawbacks, depending on both the structure of the indexed terms and the type of queries performed. As a result, many automated theorem provers use a combination of different term indices to provide a performant index for every query.

For example, Vampire uses code trees, path indexing and discrimination trees for different proof methods [17]. Other automated theorem provers, such as E [18] and SPASS [23], also take advantage of multiple term indexing techniques.

Interactive theorem provers, such as Isabelle, have traditionally placed more emphasis on trustworthiness and streamlined processes. In more recent times, large verification projects, such as the CompCert C-compiler [11] and the seL4 microkernel [9], became more prominent. Moreover, the Archive of Formal Proofs [2] has been steadily growing, currently hosting more than 165.000 theorems and 3 million lines of code. All these projects have shown the need for better proof automation.

One approach to this problem is Sledgehammer, a tool to apply automated theorem provers to goals in Isabelle. While this allows Isabelle to benefit from the work on automated theorem provers, it is faced with many hurdles. Each invocation requires the conversion of the internal representation of the goal and knowledge base to the representation of each prover and, upon success, a reconstruction of the proof in Isabelle. [6, 3]

Another approach is building general proof methods directly in Isabelle, which relies on term indexing to achieve comparable performance. So far, term indices were used only sparingly in Isabelle as most proof methods were not limited by the performance of term indices. Therefore, only an implementation of discrimination trees is provided as part of the Isabelle/ML environment. As increasingly complex proof automation is written in Isabelle's user space, more term indexing techniques are required to exploit their respective strengths.

1.2 Contributions

To address this need for more term indices, we defined a unified interface for them. This interface will simplify the implementation of additional term indices and allow users to swap one term index for another with minimal effort. Thereby, a user can choose the most performant term index for their context. The interface in its current form is limited to the functions previously implemented by the discrimination tree index but can easily be extended.

In addition to the interface, we provide an implementation of path indexing for first-order terms. Adapting path indexing from its standard representation in the literature to the more general term indexing interface defined in Isabelle/ML is the main contribution of this thesis. A major challenge was the generalisation of path indexing to store sets of values indexed by terms rather than only storing terms in an efficient manner.

To increase our confidence in the correctness of our optimised implementation we also adapted SpecCheck [5], a testing suite for Isabelle/ML inspired by QuickCheck [7]. We implemented a widely applicable term generator and used it for our tests and benchmarks. While doing so, we also modularised, documented and refactored the SpecCheck framework to increase reusability and code quality. Those changes are not discussed in this thesis. Interested readers can find the updated framework in the repository[1]. We plan to upstream the changes to the Isabelle repository in the near future.

1.3 Thesis Outline

Chapter 2 starts with the preliminaries of this thesis, including a brief overview of terms in first-order logic, λ-calculus and Isabelle/ML. We also introduce the term indexing problem. In Chapter 3, we introduce the path indexing and discrimination tree indexing methods formally. In addition, we discuss the complications faced while implementing and optimising path indexing for Isabelle/ML.

In Chapter 4 we evaluate the performance of our evaluation. Section 4.2 focusses on the effect of the optimisations and Section 4.3 on the relative performance of path indexing and discrimination trees with regards to the queries and the insertion and deletion of terms from the index. We address potential shortcomings of our evaluation in Section 4.4.

We conclude the thesis with a brief summary of our results in addition to some final thoughts on potential future developments and related work.

[1] https://gitlab.lrz.de/ga85wir/spec_check

2 Preliminaries

2.1 First-Order Logic

First-order logic is a formal language used to, amongst others, formalise reasoning, including artifical intelligence, logic programming and automated deduction systems. In this thesis we are only interested in terms. Therefore, we disregard formulas, relations and quantifiers. A more extensive introduction can be found in [1].

A symbol is either a variable, a constant or a function. We choose all variables from the infinite set $\mathcal{V} \coloneqq \{x, y, z, x_1, x_2, \dots\}$, all constants from the infinite set $\mathcal{C} \coloneqq \{a, b, c, c_1, c_2, \dots\}$ and all functions from the infinite set $\mathcal{F} \coloneqq \{f, g, h, f_1, f_2, \dots\}$. Whenever possible, we use only the first three symbols of each set for better readibility.

The arity of a symbol $\text{arity}(s)$ is a positive integer representing the number of arguments the symbol is applied to. All constants have a fixed arity of 0 while every function f has a fixed $\text{arity}(f) \geq 1$. A variable x has an arbitrary but fixed arity depending on its context.

A term in first-order logic, chosen from the infinite set $\mathcal{T} \coloneqq \{t, u, v, t_1, t_2, \dots\}$, is a symbol s applied to $\text{arity}(s)$ arguments, where each argument again is a term.

Example 2.1. Assume $\text{arity}(f) = 1$ and $\text{arity}(g) = 2$ and, for all other symbols s, $\text{arity}(s) = 0$. Then, the terms $f(a)$ and $g(f(x), a)$ are well-formed while the terms $f(a, b)$, $f(g)$ and $a(b)$ are not.

2.1.1 Generalisation and Unification

Definition 2.2. A substitution is a partial function $\rho : \mathcal{V} \longrightarrow \mathcal{T}$. We denote by $t\rho$ the term obtained by replacing all variables v in t by $v\rho$ if v is in the domain of ρ. We write $[t_1/x_1, \dots, t_n/x_n]$ for the substitution $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$.

When applying the substitution $\rho = [a/x]$ to the term $t = f(x, y)$, we get the term $f(a, y) = t\rho$. As y is not in the domain of ρ, it is not modified in the term. Note that ρ is applied only once to the term, that is, $x[y/x, a/y] = y$ even though y would be substituted by a in the original term.

Definition 2.3. Given two terms t, u, we say that t is a *generalisation* of u and u an instance or specialization of t if and only if there exists a substitution ρ such that $t\rho = u$. Similarly, we call t and u unifiable if and only if there exists a substitution ρ such that $t\rho = u\rho$. In this case, ρ is called a unifier of t and u.

Example 2.4. Let $t \coloneqq f(x)$ and $u \coloneqq f(g(a))$. Set $\rho \coloneqq [g(a)/x]$. Then $t\rho = f(g(a)) = u\rho = u$. Hence, t is a generalisation of u, t and u are unifiable and ρ is a unifier t and u.

The question of whether term t is a generalisation of term u is also known as the matching problem in the literature. Similarly, determining whether t and u are unifiable is called the unification problem. [14]

2.1.2 Variable Identity

When solving a matching or unification problem, we must pay attention to variables occuring multiple times. For example, $f(x,x)$ is not a generalisation of $f(a,b)$ as x cannot be substituted by both a and b. Similarly, $t = x$ is a generalisation of $u = f(x)$ using the substitution $\rho = [f(x)/x]$ but they are not unifiable.

Tracking substitutions for variables while solving matching problems in term indices, as is done in this thesis, complicates matters substantially. Most work in the literature and practical implementations hence simplify matters by disregarding the identity of variables. That is, they replace them by a placeholder term, which we call $*$. For example, the terms $f(x,y)$ and $f(z,z)$ are both treated as $f(*,*)$ and are therefore not differentiated.

We also employ this simplification in this thesis.

Definition 2.5. Variants are terms identical up to loss of variable identity.

Each $*$ is treated as a unique placeholder and, while solving a matching problem, we assign each $*$ a unique index. For example, the terms $f(*_1, a)$ and $f(b, *_2)$ are unifiable with unifier $\rho = [b/*_1, a/*_2]$.

2.2 Lambda Calculus

The λ-calculus is a formal language used to express computation based on functions. It is defined by a grammar for constructing λ-terms and rules for reducing them. The set of terms $\mathcal{T}$ of the untyped λ-calculus is defined as follows:

1. An infinite set $\mathcal{V}$ of variables. Each variable is a term.
2. If t is a term and x is a variable, then $\lambda x.t$ is a term. This is called an abstraction and represents a function with parameter x.
3. If t and u are terms, then tu is also a term. This is the application of the first argument to the second one.

In this thesis, we are only concerned with the lambda calculus as far as we have to model first-order terms as part of the lambda calculus-based language of Isabelle. We are hence ot be concerned with reduction rules nor types in this thesis. For a more detailed introduction, see for example [13].

2.3 Isabelle

Isabelle is a generic interactive theorem prover. By design, it uses a metalogic, called Isabelle/Pure, to embed other logics and provide a deduction framework. To do so, Isabelle/Pure uses a higher-order logic. The very basis of this metalogic are simply typed λ-terms within which theorems and inference rules are embedded. [24]

Isabelle is written for the most part in Standard ML (SML) and can also be extended at runtime. It is divided into a small kernel that verifies the correctness of all proofs and the user space within which one can axiomatise new theories and build stronger proof automation.

2.3.1 Term Representation in Isabelle

The λ-terms are a variant of simply typed λ-calculus. They are defined, with minor changes for the sake of simplicity, as follows:

```
datatype term =
    Const of string * typ
  | Free of string * typ
  | Var of string * typ
  | Bound of int
  | Abs of string * typ * term
  | $ of term * term
```

1. `Const` and `Free` both represent a fixed symbol. The latter is used to represent fixed variables in the process of a proof. In this thesis, this distinction is irrelevant: both will be treated as first-order constants.

2. `Var` represents a variable, i.e. it is a placeholder and can be replaced by an arbitrary term of the same type.

3. `Bound` is a variable bound by a lambda term encoded as a de Bruijn index [4].

4. `Abs` is an abstraction. Although Isabelle uses de Bruijn indices, variables are named for pretty printing purposes.

5. `$` represents the application of the first argument to the second one.

We will ignore the types of terms and simply assume type correctness of all given terms. For example, the λ-term $(\lambda x.\ x)\ a$ can then be represented directly as `Abs x (Bound 1) $ Const a`. The application `$` is written infix and is left-associative, i.e. $f\ x\ y$ is written as `Const f $ Var x $ Var y` whereas $f\ (g\ x)$ is written as `Const f $ (Const g $ Var x)`. As there are no tuples in this term representation, all functions are curried by default. That is, `Abs x (Abs y (Const f $ Bound 2 $ Bound 1))` represents the λ-term $(\lambda x\, y.\ f\ x\ y)$.

We can embed first-order terms in these λ-terms. Variables with an arity of 0 and constants map directly to `Var` and `Const` respectively. Likewise, a function symbol can be represented using `Const`. Terms involving functions are represented by a chain of applications of the constituent subterms. For example, the term $f(a, g(x))$ is represented by `Const f $ Const a $ (Const g $ Var x)`. Note the parentheses around $g(x)$ to differentiate this term from $f(a, g, x)$.

We assume for the sake of simplicity that every term consists of only `Const`, `Free`, `Var` and `$`. Occurrences of `Free` are treated as `Const`. `Abs` are not required for first-order terms and dangling `Bounds`, that is, indices pointing to a non-existing abstraction, are excluded, too.

2.4 Term Indexing

A term index is a data structure that allows us to efficiently store and query a set of terms. It provides, for example, a *unifiables* query that takes a term index and a term t and retrieves all terms from the term index that are unifiable with t.

Definition 2.6. A term index is an indexed set of terms $\mathcal{I}$ together with the querie operations variants(t), instances(t), generalisations(t) and unifiables(t) that return the the variants, instancesn, generalisations and unifiable terms with respect to t stored in $\mathcal{I}$, respectively. Moreover, it provides two operations insert(t) and delete(t) to insert and remove a term t from the indexed set of terms.

A term index usually shares structures of similar terms to improve its performance. For example, when retrieving unifiable terms from the set $\{f(*), f(a), f(g(a)), g(a)\}$ with the query term $g(x)$ and $f(*)$ fails to unify with $g(x)$, there is no need to also check whether $f(a)$ is a feasible candidate as it is an instance of $f(*)$.

There is a great variety of term indices and their grouping mechanisms. Furthermore, some term indices can also implement other operations efficiently. Some examples, discussed in more depth in [10], are the union of two indices and the retrieval of terms unifiable with any term in a query set.

Many specialised operations can be implemented but, alas, we cannot predict which operations will be used. As they can be emulated less efficiently by the simpler operations, we will limit ourselves to the basic query operations, retrieving all the variants, instances, generalisations and unifiables of a query term.

As mentioned in Section 2.1.2, we disregard identity of variables. By doing so, we simplify the implementation significantly but obviously obtain incorrect results when retrieving terms. To be precise, the queries will potentially return incorrect terms in addition to the correct terms.

Definition 2.7. A query returning a superset of the correct answer is called an overapproximating query. Similarly, we call a term index overapproximating if it supports only overapproximating queries.

Depending on the context, we may use this overapproximated result either directly or filter the returned overapproximation with some post-processing methods to obtain the exact set of candidates. Handling the identity of variables correctly in the term index significantly complicates the implementation and sometimes even performs worse than an overapproximative approach [10]. We hence focus on overapproximative approaches disregarding the identity of variables in this thesis.

3 Term Indexing

In the following sections we give an overview of path indexing and discrimination trees. We also take a closer look at some details of their implementation in Isabelle/ML as they differ in many places significantly from the approaches chosen in most literature.

3.1 Path Indexing

A term can be represented as a tree with all symbols s with $\text{arity}(s) = 0$ as leafs and all functions $f(x_1, \dots, x_n)$ as internal nodes with the x_i as children. Within this tree, every symbol has a position determined by the nodes traversed to reach this symbol. We represent this as a sequence of $(symbol, index)$ pairs with the index describing which argument of a function is traversed. We call this sequence a path. The path of a symbol s begins with the top symbol and ends with the index at which s is located. For example, $\langle(f, 2), (g, 1)\rangle$ is the path of the symbol a in $t = f(x, g(a, b))$. Figure 3.1 shows all the paths and symbols of t. We represent a path by enclosing the sequence of $(symbol, index)$ pairs with $\langle\rangle$.

Definition 3.1. A path is a sequence of $(symbol, index)$ pairs where the index describes the index of the next argument to traverse.[1] $Symbol_t(p)$ refers to the symbol associated with path p in the term t.

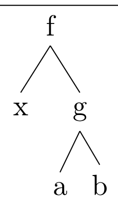
Tree Representation	Path p	$Symbol_t(p)$
	$\langle\rangle$	f
	$\langle(f, 1)\rangle$	$*$
	$\langle(f, 2)\rangle$	g
	$\langle(f, 2), (g, 1)\rangle$	a
	$\langle(f, 2), (g, 2)\rangle$	b

Figure 3.1: The paths and associated symbols of $t = f(x, g(a, b))$

A $(path, symbol)$ pair can be interpreted as a constraint on a term where at path $path$ there must be the symbol $symbol$. For example, the constraint $(\langle(f, 1)\rangle, c)$ is only fulfilled by terms of the form $f(c, \dots)$. A term gives rise to a set of $(path, symbol)$ pairs, which, when interpreted as constraints, uniquely identify this term up to loss of variable identitification.

A term t can either be represented by a set of paths and the $Symbol_t$ mapping or by listing each associated symbol explicitly in a set of $(path, symbol)$ pairs where $symbol = Symbol_t(path)$. We will choose whichever notation is clearer in the given context.

[1] This is in contrast to coordinate indexing which only uses a sequence of indices [19].

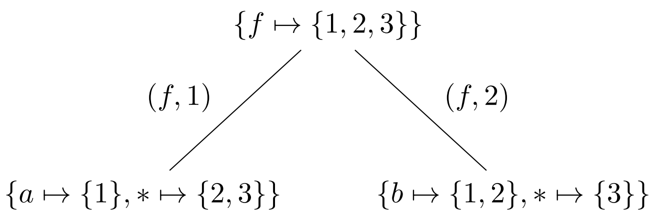
3.1.1 Structure

A path index builds on this idea of constraints and associates each $(path, symbol)$ pair with a set of terms that fulfill this constraint. For example, a path index storing the terms $\{f(x), f(a), g(a)\}$ will associate $(\langle\rangle, f)$ with the two terms $f(x)$ and $f(a)$.

Definition 3.2. A path index is a function $index : Path \times Symbol \longrightarrow 2^{Term}$ that maps a constraint $(path, symbol)$ to the set of terms that fulfill this constraint and are stored in the path index.

Storing the terms of *index* such that it can be quickly evaluated for a $(path, symbol)$ pair can be achieved in multiple ways. We decided to use a prefix-sharing tree based approach as many of the paths share prefixes. The nodes of the tree contain a function $Terms_p : Symbol \longrightarrow 2^{\mathcal{T}}$ where p is the path from the root to the node. The edges are labelled with $(symbol, index)$ pairs, which correspond to the elements of a path.

Figure 3.2 shows a path index stored as a prefix-sharing tree. Note that we only use numbers to represent terms for better readability. The root contains a mapping from the symbol f to all three terms as they all share this path. 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 other two terms.



Index	Term
1	$f(a, b)$
2	$f(x, b)$
3	$f(y, z)$

Figure 3.2: A path index storing three terms

When we insert a path p of a term t, we start at the root and traverse the tree according to p. Once we reach the end of p we extend $\text{terms}_p(\text{symbol}_t(p))$ by $\{t\}$. To insert a term we simply insert all the paths that describe this term. This requires the insertion of many similar paths which benefits from the prefix sharing. Deleting a term t is done almost identically. Instead of extending the $\text{terms}_p(\text{symbol}_t(p))$ by $\{t\}$, we remove it.

3.1.2 Queries

Queries are answered by combining the different $\text{terms}_p(s)$ sets with intersections or unions to retrieve a set of terms. For example, a variants query for the term $t = f(x, g(a, b))$ procedes as follows:

1. Compute the set of $(path, symbol)$ pairs describing the term.
2. Retrieve the corresponding $Terms_p(s)$ from the index.
3. Intersect the $Terms_p(s)$ to retrieve only the terms u containing the same symbols at identical paths as the query term, that is, $Symbol_t(p) = Symbol_u(p)$

Under the assumption of consistent typing, we retrieve only terms of identical structure as the query term. Due to the loss of variable identity we also retrieve variants of the query term in addition to the query term itself (if it is stored in the index).

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 $f(x_1, \ldots, x_n)$ is unifiable with term t if and only if $t = x$ or $t = f(y_1, \ldots, y_n)$ where, for all i, x_i is unifiable with y_i.

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. Table 3.1 shows the recursive definition for all the queries. As can be seen, the different queries are quite similar, with variants being the most restrictive and unifiables the least restrictive. $AllTerms$ is the set of all terms stored in the index and represents a wildcard at this path as intersecting $AllTerms$ with an arbitrary $terms_p$ returns $terms_p$.

Query \ Arguments	$Q(p,x)$	$Q(p,a)$	$Q(p, f(t_1, \ldots, t_n))$
$Q =$ variants	$terms_p(*)$	$terms_p(a)$	$\bigcap_i Q(\langle p, (f,i)\rangle, t_i)$
$Q =$ instances	$AllTerms$	$terms_p(a)$	$\bigcap_i Q(\langle p, (f,i)\rangle, t_i)$
$Q =$ generalisations	$terms_p(*)$	$terms_p(a) \cup terms_p(*)$	$\bigcap_i Q(\langle p, (f,i)\rangle, t_i) \cup terms_p(*)$
$Q =$ unifiables	$AllTerms$	$terms_p(a) \cup terms_p(*)$	$\bigcap_i Q(\langle p, (f,i)\rangle, t_i) \cup terms_p(*)$

Table 3.1: The recursive definition of the queries

3.2 Discrimination Tree

A discrimination tree index, also known as discrimination net index, is a prefix-sharing tree, similar to a trie, which stores the indexed terms. To determine the leaf at which a term is stored we use the preorder traversal of the term. It is obtained by simply reading the written term from left to right. For example, the preorder traversal of $t = f(c, g(x, y))$ is $\langle f, c, g, x, y\rangle$. Since we disregard variable identities, this will further be simplified to $\langle f, c, g, *, *\rangle$.

Definition 3.3. $Preorder(t)$ is the sequence of symbols obtained by the preorder traversal of the term t. For symbols s with arity$(s) = 0$ it is the symbol s itself. The preorder traversal of a function $f(x_1, \ldots, x_n)$ is $\langle f, Preorder(x_1), \ldots, Preorder(x_n)\rangle$ For the sake of simplicity, we flatten the sequence, e.g. $\langle f, \langle g, x\rangle\rangle$ becomes $\langle f, g, x\rangle$.

3.2.1 Structure

We store the mapping $Preorder(t) \mapsto t$ in the prefix-sharing tree. The symbols in the $Preorder(t)$ sequence are the labels of the edges leading to a leaf where t is stored. Internal nodes store no information. $Preorder(t)$ always addresses a leaf as, under the assumption of type consistency, it is impossible for $Preorder(t)$ to be a prefix of $Preorder(u)$ if $t \neq u$. A discrimination tree storing multiple terms can be seen in Figure 3.3. As can be seen, the common prefix f of all terms is shared in memory. On the other hand, the common postfix b is not shared.

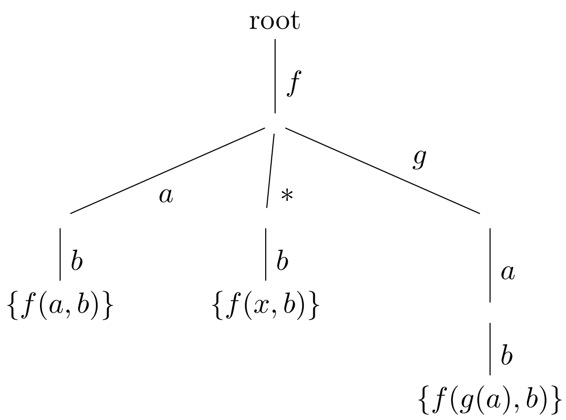


Figure 3.3: A discrimination tree index storing three terms

Insertion and deletion are straightforward in discrimination tree indexing. When inserting a term t, we traverse the tree according to $Preorder(t)$, reaching a leaf, and insert t into the set at the leaf. Deletion works identically, except we remove the term from the set.

3.2.2 Queries

The queries are implemented as a recursive algorithm on the nodes of the tree and $Preorder(t)$ of the query term t. Starting at the root, we traverse the tree by selecting the child node corresponding to the first symbol of $Preorder(t)$. We recursively continue at the child node while removing the first symbol from the sequence. For example, a variants query for $f(y,b)$ on the index in Figure 3.3 would traverse the tree by following the edges $\langle f, *, b\rangle$, retrieving the term $f(x,b)$

Definition 3.4. $\mathrm{slp}(N, s)$ is the symbol lookup operation. It takes a node N of the discrimination tree and a symbol s, returning the child node of N reached by following the edge labelled with symbol s. If no such node exists, we return an empty node with no children. We write repeated applications of slp, such as $\mathrm{slp}(\mathrm{slp}(\mathrm{slp}(N,a),b),c)$, as $\mathrm{slp}(N, \langle a,b,c\rangle)$

Definition 3.5. $terms(N)$ retrieves the terms stored in N. If N is not a leaf, we return the empty set.

slp and *terms* can both be implemented very efficiently and using them, we can write the variants query as $terms(\mathrm{slp}(root, \langle f, *, b\rangle))$. Unfortunately, the other queries are more intricate as they may replace variables by arbitrary terms or vice versa, with unification allowing both.

For every constant symbol in the term of a generalisations or unifiables query, we form the union of both the query on the node $\mathrm{slp}(N, c)$ as well as $\mathrm{slp}(N, *)$. This ensures that indexed terms containing variables instead of constants are also retrieved. For example, the unifiables of $f(a, b)$ are retrieved by forming the union of $\mathit{terms}(\mathrm{slp}(M, \langle a, b\rangle))$ and $\mathit{terms}(\mathrm{slp}(M, \langle *, b\rangle))$ where $M = \mathrm{slp}(\mathit{root}, f)$, in addition to the empty sets of $\mathit{terms}(\mathrm{slp}(\mathit{root}, *))$ and $\mathit{terms}(\mathrm{slp}(\mathit{root}, \langle f, a, *\rangle))$.

A variable in the instances or unifiables query term must also be handled differently. As the variable can be substituted arbitrarily, we continue the query at every child of the current node, taking the union of the retrieved terms. For example, all the terms stored in Figure 3.3 are instances of $f(*, b)$. We notice that at the node $M = \mathrm{slp}(\mathit{root}, f)$, we must continue in every branch.

But what about functions? The variable may be replaced by terms with an arbitrary number of arguments. We must skip the nodes corresponding not only to g but also each argument $x_1, \ldots, x_n$ of $g(x_1, \ldots, x_n)$. When continuing in the g branch, we must not continue at $\mathrm{slp}(M, g)$, but at $\mathrm{slp}(M, \langle g, a\rangle)$, that is, we need to *skip* not only g but $\langle g, a\rangle$ as the variable is substituted by $g(a)$. Figure 3.4 shows the nodes reached by skipping a subterm after reaching the node $\mathit{slp}(\mathit{root}, f)$, i.e. $\mathit{skip}(\mathit{slp}(\mathit{root}, f))$

Definition 3.6. $\mathit{skip}(N)$ returns the set of nodes obtained by skipping a single term starting at N. That is, for a constant c with $\mathrm{arity}(c) = 0$ we return $\mathrm{slp}(N, c)$ (which is a direct child of N). For a function $f(x_1, \ldots, x_n)$, we return the nodes $\mathit{skip}^n(\mathrm{slp}(N, f))$. If for all x_i $\mathit{arity}(x_i) = 0$, skip^n retrieve the nodes $1 + n$ levels below N.

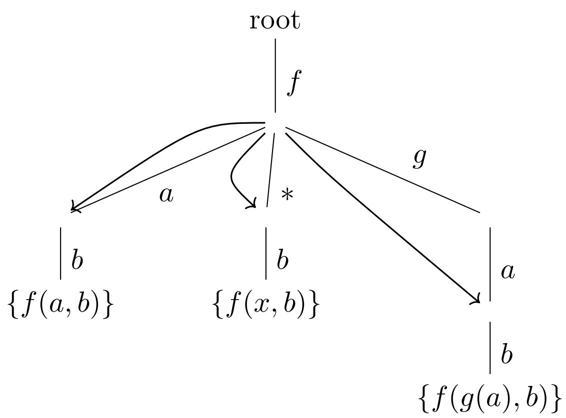


Figure 3.4: The Nodes in $\mathit{skip}(\mathit{slp}(\mathit{root}, f))$

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 Table 3.2. The base case of $Q(N, \langle\rangle) = \mathit{terms}(N)$ is identical for all queries and not included in the table. 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.

Revisit query table. Looks odd, especially at the bottom

Query \ Arguments	$Q(N, \langle x, t_2, ..., t_n \rangle)$	$Q(N, \langle a, t_2, ..., t_n \rangle)$	$Q(N, \langle f(x_1, \ldots, x_n), t_2, ..., t_n \rangle)$
$Q =$ variants	$Q(\mathrm{slp}(N, *), \langle t_2, ..., t_n \rangle)$	$Q(\mathrm{slp}(N, a), \langle t_2, ..., t_n \rangle)$	$Q(\mathrm{slp}(N, f), \langle x_1, \ldots, x_n, t_2, ..., t_n \rangle)$
$Q =$ instances	$\bigcup_{M \in \mathit{Skip}(N)} Q(M, \langle t_2, ..., t_n \rangle)$	$Q(\mathrm{slp}(N, a), \langle t_2, ..., t_n \rangle)$	$Q(\mathrm{slp}(N, f), \langle x_1, \ldots, x_n, t_2, ..., t_n \rangle)$
$Q =$ generalisations	$Q(\mathrm{slp}(N, *), \langle t_2, ..., t_n \rangle)$	$Q(\mathrm{slp}(N, a), \langle t_2, ..., t_n \rangle)$ $\cup\, Q(\mathrm{slp}(N, *), \langle t_2, ..., t_n \rangle)$	$Q(\mathrm{slp}(N, f), \langle x_1, \ldots, x_n, t_2, ..., t_n \rangle)$ $\cup\, Q(\mathrm{slp}(N, *), \langle t_2, ..., t_n \rangle)$
$Q =$ unifiables	$\bigcup_{M \in \mathit{Skip}(N)} Q(M, \langle t_2, ..., t_n \rangle)$	$Q(\mathrm{slp}(N, a), \langle t_2, ..., t_n \rangle)$ $\cup\, Q(\mathrm{slp}(N, *), \langle t_2, ..., t_n \rangle)$	$Q(\mathrm{slp}(N, f), \langle x_1, \ldots, x_n, t_2, ..., t_n \rangle)$ $\cup\, Q(\mathrm{slp}(N, *), \langle t_2, ..., t_n \rangle)$

Table 3.2: The recursive definition of the queries

3.3 Term Indexing in Isabelle/ML

As Isabelle has now been used for over 30 years, a number of data structures have already been implemented to store terms. One of the simplest approaches is the `termtable` [2], a balanced 2-3 tree, storing terms and differentiating them on all attributes, namely their structure, symbols and types. Therefore, this approach is best used when an exact lookup is necessary. On the other hand, `termtable` does not offer any support for the more complex queries such as instances or unifiables.

Another data structure present in Isabelle is the `discrimination tree` [3]. Despite being based on the concepts introduced above, the discrimination tree implementation in Isabelle/ML stores arbitrary sets of values indexed by terms. This is useful when we want to tag terms with multiple attributes, for example, introduction and simplification rules.

The interface of the discrimination tree is mostly identical to the one introduced in Section 2.4. The queries return sets of values instead of sets of terms and insertion and deletion use key-value pairs, similar to hash tables. To modify the stored values, we use a term to address a leaf and insert or delete values from the respective value set. In addition, the index raises an exception if it detects duplicate key-value pairs. The value comparison used for the detection of duplicates is supplied by the user and can, therefore, also be the constant function returning false, i.e. `eq(v1,v2) = false`.

3.3.1 Caveats of current Implementation

The generalisation of storing terms to storing arbitrary sets of values is relatively simple for discrimination trees. Each leaf is addressed by only one preorder traversal and therefore stores only variants of one term. As such, we can simply replace this set of terms with a set of arbitrary values.

When inserting or deleting a value for a term t, $\mathit{preorder}(t)$ is used as the key to address the node where the value should be stored at. This results in some potentially surprising behaviour. We illustrate this with some examples. We write (t, v) for term-value pair

[2] https://isabelle-dev.sketis.net/source/isabelle/browse/default/src/Pure/term_ord.ML;89cf7c903acad179e40c10f2f6643d3c21448f47$226

[3] https://isabelle-dev.sketis.net/source/isabelle/browse/default/src/Pure/net.ML;89cf7c903acad179e40c10f2f6643d3c21448f47

stored and DT for the (initially) empty discrimination tree.

1. Inserting $(a, true)$ and $(b, true)$ into DT stores $true$ at both a and b. Retrieving the unifiables of x returns the multiset $\{true, true\}$ as both a and b are unifiable and the queries do not deduplicate the results.
2. Inserting $(x, true)$ and $(y, true)$ into DT results in an exception as both are stored in the same node of the tree and the values are identical.
3. Similarly, deleting $(y, true)$ after inserting $(x, true)$ into DT deletes the value.
4. Inserting (x, x) and (y, y) into DT stores both variables x and y in the same node as the values are different.
5. After inserting $(x, true)$ into DT, we cannot delete this value without knowing the term used to address the node where the value is stored.

The lack of deduplication in queries is necessary as the insertion of an identical value at different nodes is valid. Therefore, the different instances of the value should be treated separately. Items 2 to 5 may not seem too surprising when the user keeps in mind that the discrimination tree stores key-value pairs and the terms used as keys disregard the identity of variables. On the other hand, terms stored as values, generally, will respect variable identity as it may be relevant in the user's context. Nevertheless, the user must be wary to pay attention to these potential pitfalls.

3.3.2 Adapting Path Indexing

Unfortunately, most literature [19, 14, 22] on path indexing only covers the storage of terms. Reproducing the behaviour of the discrimination tree implementation correctly and efficiently takes some effort. The queries on a path index rely primarily on the intersection of sets of terms as every function results in a number of intersections.

We recall that a term is never explicitly stored in path indexing. Instead we represent a term by a collection of paths, each storing the set of terms containing this path. Naively replacing this set of terms by a set of values does not work as we can no longer detect duplicates and handle deletions correctly. For example, a path index storing the key-value pairs $(f(a, *), true)$ and $(f(*, b), true)$ has the value $true$ stored at the $(p, Symbol_t(p))$ pairs $(\langle\rangle, f)$, $(\langle(f, 1)\rangle, a)$ and $(\langle(f, 2)\rangle, b)$, amongst others. When inserting $(f(a, b), true)$, it is impossible to determine whether this key-value pair has already been inserted before.

Therefore, we must also store the key of a value at each path. Doing so allows us to reproduce the behaviour of the discrimination tree in the path index. Note that we need only check if one path, for example the top symbol, stores the same term-value pair on insertion. If one path does not contain the same key-value pair, no other path will. Nevertheless, some optimisations can still be made.

3.3.3 Combining Path Indexing and Termtables

A potential problem remains in the above approach. Insertion is fast because we need to only compare one path to determine whether an identical key-value pair has already been inserted. Deletion of (t, v), on the other hand, requires us to remove (t, v) from the set at every path which necessitates repeatedly comparing (t, v) with the other key-value pairs stored. When we have many terms that share their prefix, this overhead can become problematic as term comparison of similar terms is relatively slow.

Figure 3.5 shows a path index storing values with three similar terms as keys. For the sake of simplicity, only the key is shown. Deleting the values at term $f(g(a))$ requires two comparisons with each of $f(g(b))$ and $f(g(c))$, as they share the constraints $(\langle\rangle, f)$ and $(\langle\rangle, g)$. If the terms shared more symbols, e.g. in the form of $f(a, b, c, g(a, b, c, h(*)))$, this problem would be even more pronounced. Similarly, deleting a single value from a term associated with multiple values requires repeated comparisons of the values. As the comparison for values is supplied by the user, it may be arbitrarily slow, e.g. when storing large lists.

$$\begin{tikzcd}
\{f \mapsto \{f(g(a)), f(g(b)), f(g(c))\}\} \arrow[d, dash, "{(f,1)}"'] \\
\{g \mapsto \{f(g(a)), f(g(b)), f(g(c))\}\} \arrow[d, dash, "{(g,1)}"'] \\
\{a \mapsto \{f(g(a))\}, b \mapsto \{f(g(b))\}, c \mapsto \{f(g(c))\}\}
\end{tikzcd}$$

Figure 3.5: A path index sharing all paths

We can optimize this challenge: Instead of storing term-value pairs, we generate unique integer identifiers for each such pair. We then store those identifiers along with their associated value in the path sets. Doing so allows us to use quick integer comparison methods and to avoid repeatedly comparing terms and values.

To associate each term-value pair (t, v) with a unique identifier id, we use a `termtable`. In the termtable, we use t as the key and store the (id, v) pair. Upon insertion, we first check the termtable for an identical v stored at t, ignoring the identifier of the value. If no duplicate is found, we insert (id, v) at key t. Inserting the (id, v) pair into the tree is straightforward as we already determined that no duplicate exists.

Deletion of (t, v) works similar. By looking up t in the termtable, we obtain a set of $(id, value)$ pairs. From this list we determine the id of v, removing the (id, v) pair from the termtable in the process. Traversing the tree is, again, straightforward as we only need to compare the identifier.

This approach offers another benefit. By using the exact lookup of `termtable`, we can improve the duplicate detection to no longer ignore variable identities. In addition, we can also provide an exact lookup operation (syntactic equality) by using only the associated termtable. This reduces the overhead in applications where both the queries and an exact lookup are required. In the current implementation we use a standard `termtable` and therefore do not exactly replicate the behaviour of the discrimination tree. Switching to a variant of the `termtable` that does not respect identity of variables is fairly trivial.

3.3.4 Further Optimisations

The performance of path indexing relies on fast set operations as every function requires an intersection of the sets retrieved from the arguments. In the previous section, we already reduced the comparison for intersections to integer comparisons. By using ordered lists, provided by `OrdList` in Isabelle/ML [4], to implement the sets, we can further speed up the set operations.

When tasked to compute the intersection of two ordered lists, we need only compare the first element of each list. If they are different, we can discard the smaller value and continue. If they are equal, we know that the value is in the intersection and continue with the next element of the lists.

To improve the cache usage of the queries, we lazily evaluate the set operations. While traversing the tree, we build a tree of the required set operations, where the leafs represent the sets of values and internal nodes represent the intersection or union of a number of children. Once we have traversed the complete path index, we evaluate all the operations at once. This improves cache usage as the result of one operation can be immediately used again instead of being evicted from the cache during the traversal of the path index.

This delayed evaluation also simplifies handling the $AllTerms$ case of instances and unifiables separately. As $AllTerms$ represents a wildcard, we need not replace it by the set of all indexed term, unless it is the only relevant value. Instead, we simply disregard this set in the intersection. For example, the intersection of the three sets $AllTerms$, $\{f(a,b), f(x), g(a)\}$ and $\{g(a), g(f(a,b))\}$ is identical to the intersection of only the latter two sets.

Figure 3.6 shows a path index storing two terms and the operations tree built by a instances$(f(a,y))$ query. As y is a variable, it can be replaced arbitrarily, represented by $AllTerms$. We can then simplify the tree by removing $AllTerms$ from the intersection tree.

Index	Term
1	$f(a,b)$
2	$f(x,b)$

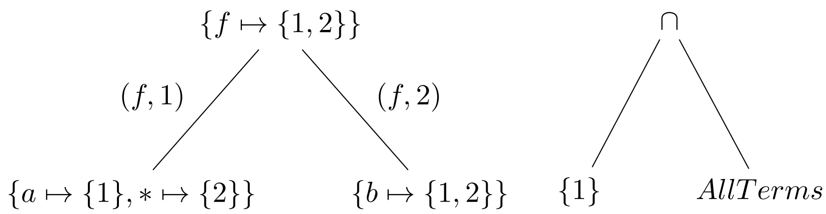


Figure 3.6: The operations tree for instances of $f(a,y)$

We attempted to further speed up the set operations by implementing a more efficient intersection operating on a larger number of children. The first idea was to start with the smallest set, thereby ensuring that less comparisons are necessary. For example,

[4] https://isabelle-dev.sketis.net/source/isabelle/browse/default/src/Pure/General/ord_list.ML;89cf7c903acad179e40c10f2f6643d3c21448f47

when intersecting the sets $\{1, 2, 3, 4\}$ $\{2, 3, 4, 5\}$ and $\{5\}$, we can start with the last set, immediately discarding all values from the first set and returning the empty set.

The second idea was to compare the head of each list before moving on to the next element instead of intersecting the first two lists completely before moving on to the third list. For example, intersecting the sets $\{1, 2, 3, 4\}$ $\{1, 2, 3, 4, 5\}$ and $\{5\}$ this way results in the values 1 through 4 being discarded directly. While they are present in the first two lists, they are smaller than 5. This is opposed to the naive version, in which we build the intermediate result $\{1, 2, 3, 4\}$ before moving on to the last list.

Unfortunately, both ideas proved to be slower. Due to the linked lists used by `OrdList` providing no efficient `length` function, the first idea resulted in significant overhead. The second idea was, unfortunately, also slower although we could not determine the exact reason. Perhaps the elements of a list are allocated, at least piecewise, consecutively in memory. In this case, accessing only one element of each list would be detrimental to an efficient usage of the cache.

Although we repeatedly store identical values at different locations in the path index, this does not impact memory consumption. Isabelle/ML is based on the Poly/ML runtime [16] which provides data sharing. This results in copies of immutable values requiring almost no additional memory. A related feature is `pointer_eq` [20] which makes use of the data sharing mechanism and compares immutable values quickly. Unfortunately, we cannot directly take advantage of this as the user may wish to use a constant function returning false, i.e. `eq(v1,v2) = false`, for comparison.

Some of the optimisations proposed in this section have also been described in Section 5.3.1 of [21]. The authors there propose some further optimisations for the computation of intersections based on database techniques and using references. However, as the usage of references is discouraged in Isabelle/ML, we did not investigate the latter any further.

4 Evaluation

The path index implemented as part of this thesis competes directly with the discrimination tree implementation already present in Isabelle/ML. Furthermore, the combination of path indexing with termtables increases complexity and overhead. In this chapter, we will evaluate the impact of the termtables and compare the performance of the discrimination tree with the path index.

4.1 Approach

For the experiments, randomly generated term sets are used. During generation, the terms are represented as a tree with functions as internal nodes with the children representing the arguments. This is later converted to the applicative style used in Isabelle/ML. Starting at the root node, a random number of arguments is chosen. We then recursively descend into each argument, generating another random number of arguments. This number ranges from 0 to 4. As a result, 20% of the generated nodes are nullary constants and the remaining 80% are functions with 1 to 4 arguments. In addition, the depth of the tree is limited to 6, that is, every node on the sixth level is a constant.

Each generated term set has an associated variable frequency $f \in \{0.00, 0.01, 0.03, 0.1\}$. f specifies the percentage of symbols s that are selected from the set of variables $\mathcal{V}$. A function g may also be a variable, which is, strictly speaking, not allowed in first-order terms. In this case, both term index implementations treat the subtrem as a variable. Therefore, a higher variable frequency effectively decreases the size of the term.

Example 4.1. Assume $f = 0.1$ and $t = g(s)$ is a randomly generated term. Then, g and s each have a 10% chance to be variables. If g is a variable, $preorder(t) = \langle * \rangle$ and the only path of t is $\langle\rangle$ with $symbol_t(\langle\rangle) = *$.

To allow symbols to occur multiple times, we select each symbol's name at random from a finite set of names $\mathcal{N}$. The cardinality of $\mathcal{N}$ is identical to the cardinality of the term set. For example, in a term set with 10 terms, a total of 10 symbols are available. This ensures that we maintain a similar frequency of repeated occurences across different numbers of terms. Note that each term consists of many symbols, thereby ensuring that most symbols occur multiple times, either in the same term or across different terms.

While the terms generated in this way do not accurately represent terms of real applications, they allow us to efficiently stress-test the term indices. As they are randomly generated we can generate term sets of arbitrary size and test each size multiple times with different seeds to average the impact of any single term on the performance.

For the benchmarks, we generated term sets with sizes ranging from 10 to 5000. The smaller sets were tested more often to obtain test runtimes significantly larger than time measurement errors. For sizes $s \leq 100$, we generated 5000 different term sets. For sizes $100 < s < 1000$, we generated 500 sets. To restrict the runtime of the benchmarks, only 50 sets were generated for sizes $s \geq 1000$.

4.2 Combining Path Indexing and Termtables

In Section 3.3.3, we discussed the repeated comparisons necessary during insertion and deletion. By introducing unique identifiers for each $(term, value)$ pair and using termtables we reduced the time spent on comparisons, both for insertion and deletion.

To evaluate the performance of the insert operation for a term set $\mathcal{T}$, we generate a term set $\mathcal{U}$ containing an identical number of new terms, that is, $|\mathcal{T}| = |\mathcal{U}|$ and $\mathcal{T} \cap \mathcal{U} = \emptyset$. After creating a term index for $\mathcal{T}$, we start the time measurement and insert every term $u \in \mathcal{U}$ into $\mathcal{T}$, effectively doubling the size of $\mathcal{T}$. To evaluate the deletion, we also create a term index for $\mathcal{T}$ but instead delete every term $t \in \mathcal{T}$, effectively removing every indexed term.

While this approach does not accurately evaluate the performance of insertion and deletion at a given index size, it enables us to measure a long-running operation without interruptions. Instead of discarding an index after inserting a small number of terms and reusing the unmodified index, we opt to consecutively insert all terms into the same index. This ensures that infrequent but expensive operations, like the rebalancing of the 2-3 tree used by the termtables, are amortised correctly.

Despite this, we encounter some outliers in Figures 4.1a and 4.1b as well as most other graphs, predominantly at the index sizes 100 and 700. We discuss potential causes in Section 4.4.

As mentionend in the previous chapter, we require a $(term, value)$ to id mapping to speed up deletion. Upon deleting a $(term, value)$ pair, we need to remove it from the term sets of every path of $term$. Comparing the terms and values repeatedly is prohibitevily expensive. Therefore, we compare two variants using identifiers.

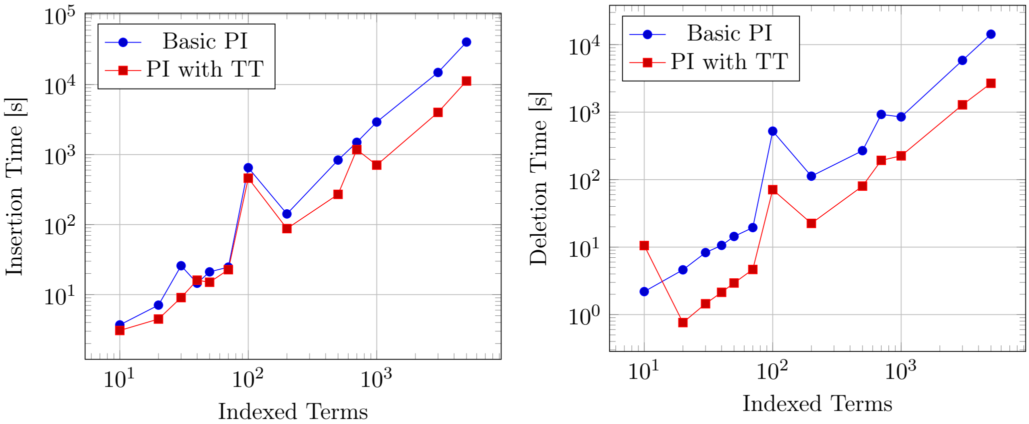
The first, basic path indexing (Basic PI) uses a linked list to associate each $(term, value)$ pair with an id. To detect duplicates, we use a variants query on the tree as this avoids an expensive linear search in the list. For deletion, we cannot avoid searching the list to retrieve the identifiers that must be deleted. The second variant, path indexing with termtables (PI with TT), uses a termtable to store $(term, value) \mapsto id$. We also use this termtable to detect duplicates and retrieve the identifiers that are deleted.

In Figure 4.1a, we can see a comparison of the respective insertion performance. Basic path indexing uses a variants query to detect duplicates of a $(term, value)$ pair. Note that this traverses all the paths of $term$ and is therefore relatively inefficient. Despite this, it can be used as a baseline for the path indexing with termtables.

It appears that combining path indexing with termtables does not significantly increase the runtime of insertion despite inserting every pair into both the termtable and the path index. While it is faster when compared with basic path indexing, we can expect that duplication detection using only the top symbol is significantly more performant. Unfortunately, this is not an option for us as the path index only stores $(id, value)$ pairs and we require the mapping of $(term, value)$ to id.

For deletion, we can see in Figure 4.1b that using a termtable is consistently and significantly faster.

Finish



(a) Insertion Performance

(b) Deletion Performance

4.3 Path Indexing and Discrimination Trees

To test the queries for a term set $\mathcal{T}$, we create both term indices PI and DT for the set $\mathcal{T}$. We execute one query for each term stored in the index. This ensures that each indexed term is retrieved at least once.

We use $t \in \mathcal{T}$ directly as the query term for the variants and generalisations query. For the instances and unifiables query, we do not use the term $t \in \mathcal{T}$ directly. Instead, we generate a generalisation of t by replacing a randomly chosen constant or function in t by a variable.

Although we can generate an instance of t to more accurately represent a generalisation query, we avoid this additional complexity. The performance of the query is nearly identical for constants and variables in the path index and the discrimination tree index. Both introduce only a single union when compared to the handling of variables. As the number of variables is relatively low, we neglect this effect. In contrast, the performance of the instances and unifiables queries on the discrimination tree index is heavily impacted by each variable occurence. See Tables 3.1 and 3.2 for reference.

Figure 4.2 shows an overview of the queries, summing the test results of each variable frequency and term sets size combination. This, of course, is not representative for the smaller sized term indices and is only used to give an intuition for the expected performance.

As we can see, path indexing performs significantly better at retrieving instances and unifiables. However, it performs poorer for generalisation queries. In addition, the variations across the different types of queries are relatively low, with the generalisations query taking about twice as long as the instances query.

The reason for the low variance in performance is found in Table 3.1. The variants and instances queries differ only in the handling of a variable. While variants retrieves the, likely small, term set from this path, the variable is simply disregarded for instances. As a result, performance of these queries is near identical.

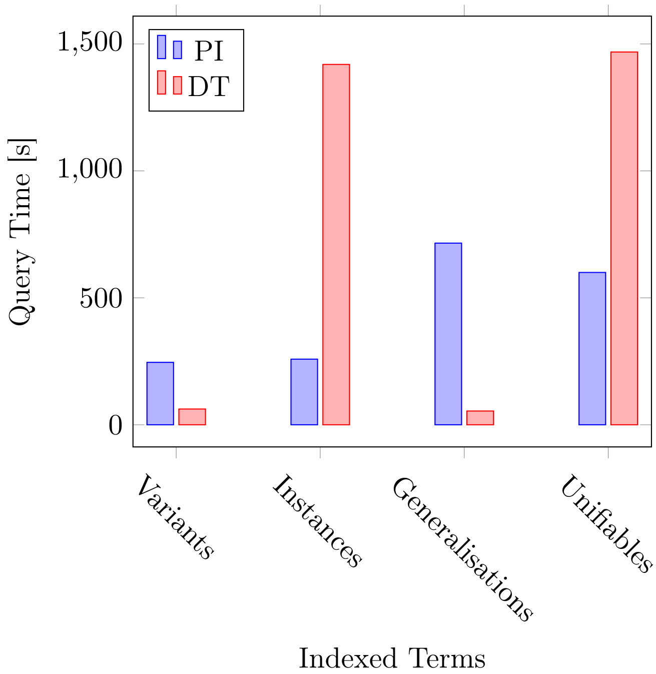


Figure 4.2: Overview of Queries

The generalisations query differs from variants by retrieving the union of two path sets for each constant or function. As each term consists mostly of constants and functions, this overhead impacts performance significantly. Unifiables suffers from the same performance problems as generalisations but reduces the number of intersections required due to variables being treated as a wildcard.

The queries on the discrimination tree index, in contrast, vary greatly in their performance. Retrieving variants is extremely fast as we only need to use the preorder traversal of the query term to reach a single leaf. Generalisations add a union at every constant or function. As these sets are not used for intersections, these unions do not significantly impact the performance. As the variants implementation computes the preorder traversal of the term instead of traversing the term directly, the generalisations query is in fact faster although this could easily be optimised.

In comparison, the instances and unifiables query are extremely slow. Both rely on the *skip* function to compute the set of nodes reached by skipping one subterm. Evaluating this function is slow as it must both traverse every child of the current node and potentially skip many nodes if a large term is skipped.

Example 4.2. Consider the set of indexed terms $\mathcal{T} = \{t_1 = f(a,x), t_2 = f(b,x), t_3 = f(c,x), t_4 = f(g(a,b),x)\}$ and the query term $u = f(x,y)$. The instances query will first lookup f and reach node $N = \text{slp}(\mathit{root}, f)$. As the next symbol of $\mathit{preorder}(u)$ is $*$, we compute $\mathit{skip}(N)$. As every indexed term shares the prefix f with u, we retrieve the set of nodes $\{\text{slp}(N,a), \text{slp}(N,b), \text{slp}(N,c), \text{slp}(\text{slp}(\text{slp}(N,g),a),b)\}$. Each of these nodes is evaluated recursively. If more terms shared some prefix with u or the subterms were larger, this evaluation will be even slower.

4.3.1 Variants

To evaluate the performance of the variants query, we test differently sized sets of terms. As variables in the term are handled identically to constants, the variable frequency of the terms is not plotted separately. We confirmed that the variable frequency had practically no impact. We average the tests of a given size with the different frequencies to reduce noise.

Figure 4.3 shows the variants query of the term indices over differently sized sets of indexed terms. Note that we run one query for each term in the set. Therefore, we expect to see a linear plot with a slope of 1 if the query performance of a term index is independent of the number of indexed terms.

In theory, only the discrimination tree index should handle the variants query as fast for large number of indexed terms as the query only relies on the preorder traversal of the query term and therefore does not interact with terms unrelated to it. The path index, on the other hand, relies on the intersection of term sets that may contain many unrelated terms and the termtables must traverse a larger 2-3 tree to reach the desired leaf. In practice, these deficits do not meaningfully affect the performance for realistically sized term indices.

Note that the variants query of path indexing can be supplemented by the exact lookup provided by the termtables. If the exact lookup of terms is sufficient, or even required, this provides a significantly faster alternative. Adapating the termtables to ignore the variable identity should also retain almost identical performance characterstics. However, due to a lack of time, we did not verify this.

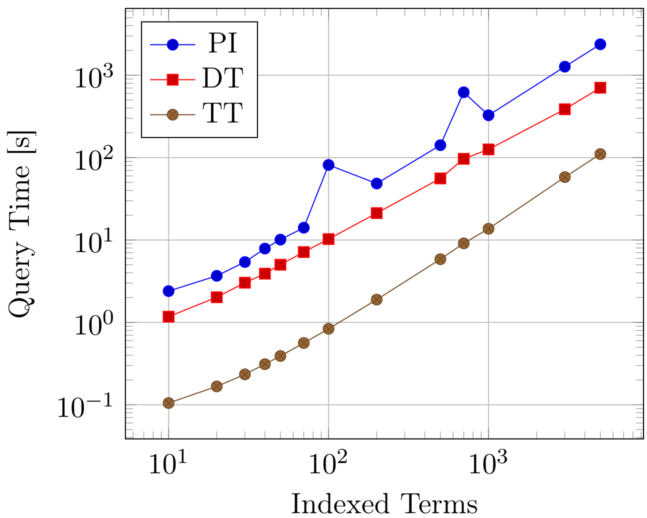
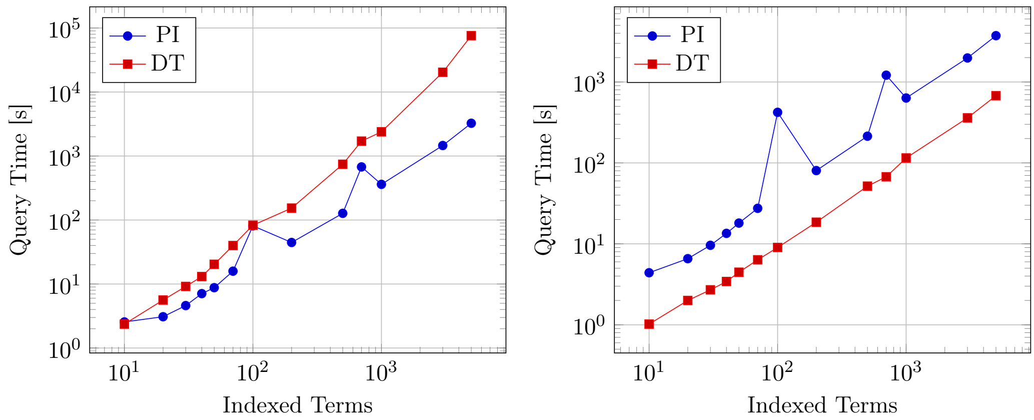


Figure 4.3: Variants Query

4.3.2 Instances and Generalisations

As already shown in the overview, the performance difference of the queries for instances and generalisations are drastic for the indices. Figure 4.4a shows that the path index dominates the discrimination tree index for all sizes when querying for instances, although this difference is more pronounced for larger indices. Similiarly, Figure 4.4b shows that the discrimination tree index is consistently and significantly faster than the path index for generalisations.



(a) Instances Query (b) Generalisations Query

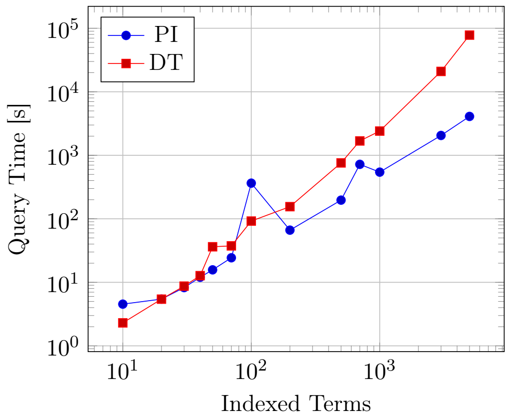
4.3.3 Unifiables

The unifiables query is likely one of the most important queries due to the wide range of applications. It is also the query in which the term indices differ the least in performance. We can see in Figure 4.5a that path indexing handles increased index sizes well. While the additional terms increase the average size of the term sets stored at the paths located close to the root, the path lists likely remain very small as it is unlikely for two large terms to share not only a constant or variable, but also all the functions leading to this symbol.

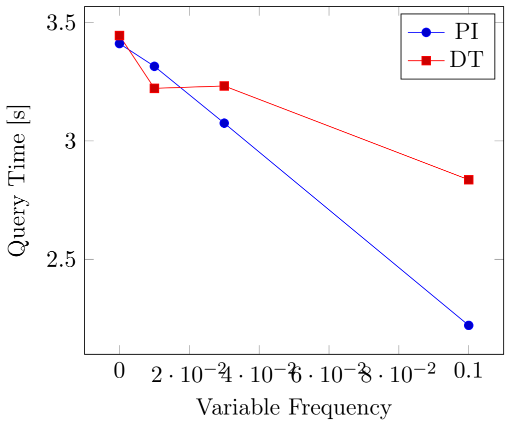
For discrimination tree indexing, on the other hand, every term sharing a prefix with the query term potentially leads to additional recursive calls due to the *skip* function returning more nodes. Note that, due to the double-logarithmic scale, the performance difference is more drastic than it appears. Increasing the number of indexed terms from 3000 to 5000, less than doubling, leads to an almost four times longer evaluation for the unifiables query on the discrimination tree.

Despite this, the discrimination tree index is comparable, or even faster, at smaller index sizes. Comparing the performance with differing variable frequencies at a fixed size of 40, as can be seen in Figure 4.5b, shows that, at lower variable frequencies, the performance is comparable.

Due to the variables replacing not only constants but also functions and both indices disregarding the arguments of a variable, increasing the variable frequency effectively decreases the size of the terms. As a result, we expect both term indices to improve with increasing variable frequencies. Path indexing, which benefits not only from the decreased size but also from the occurence of variables by treating them as wildcards, significantly improves. The discrimination tree, on the other hand, benefits only from the decreased term size but requires additional *skip* evaluations. Therefore, it improves relatively slowly and is only comparable at lower frequencies.



(a) Unifiables Query



(b) Unifiables Query at 40 Terms

4.3.4 Modifying Operations

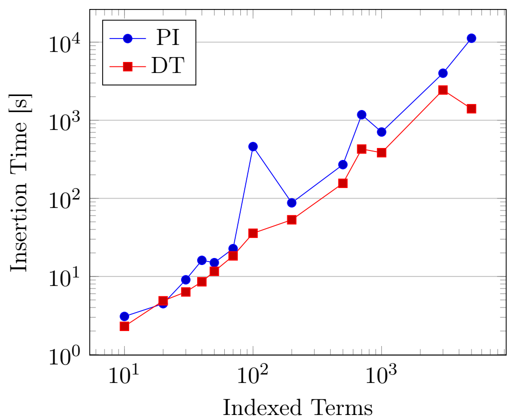
While the query performance is most important for a term index, the time spent on creation and modification of term indices may be significant if they are short-lived indices. The performance of modifying the set of indexed terms is comparable for both indices and the difference should not be a deciding factor unless an index storing multiple thousand terms must be modified frequently.

Figure 4.6a shows the insertion time for different term sizes. Again, we average the results from sets of different variable frequencies as the impact of variables is neglegible. While path indexing performs similarly well for smaller indices, it scales worse than discrimination tree indexing. This is expected as its duplicate detection is more expensive than it is for the discrimination tree. In addition, we must insert each term-value pair twice, once into the termtable and once into the tree.

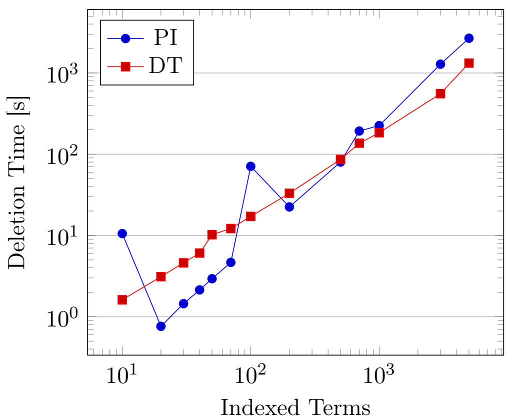
The difference is even more pronounced for deletion, as can be seen in Figure 4.6b. At small sizes, the term sets are generally small, even for the top symbol. As the number of indexed terms increases, so does the average size of the term sets. Despite the fast comparison with identifiers, this deletion must be repeated for every single path of a term. Naturally, deleting the value from potentially hundreds of increasingly large term sets increases deletion time significantly.

4.4 Shortcomings

In the evaluation of the term indices, two problems became apparent. Firstly, the term generators do not accurately represent terms of real applications. Therefore, the results



(a) Insertion Performance



(b) Deletion Performance

shown here must be taken with a grain of salt and should be compared to tests on real data for contexts that rely on high performance.

Secondly, the results from the tests show some significant and consistent outliers. These occur almost exclusively at the tests with an index size of 100 and 700. We recall from Section 4.1 that we repeat the smaller sized tests more often to get better test results. For sizes up to 100, we repeat the tests 5000 times and for sizes up to 700 we repeat them 500 times. These values were chosen in pretests to limit the runtime of the benchmark while still testing each size an appropriate number of times.

As a result, we store a total of 500000 terms for size 100 and 350000 terms for size 700 in memory during a single test run. The 5000 repetitions for size 70 also result in 350000 terms. Each of these three tests contain more terms than the other tests. The outliers occur consistently at sizes 100 and 700 but, surprisingly, size 70 is unaffected. This may be related to the overhead of the term indices.

Therefore, the first assumed cause is the exhaustion of memory as the swapping of any, even unrelated, memory will significantly slow down the benchmark. We determined that this is not the cause as the tests were run on a machine with 128 gigabytes of memory and pretests showed that memory consumption should not exceed 10 gigabytes, a fraction of the available memory.

Nevertheless, as the outliers only appear at these specific sizes, a memory related cause is likely. As these issues are highly unlikely to be caused on the level of the operating system, the next higher level may be responsible. The Isabelle/ML code is run using the Poly/ML runtime. As the Poly/ML runtime has many advanced features, like garbage collection, a data sharing mechanism for immutable values and implicit parallelism, it may be the culprit. We attempted to minimise these issues by triggering a full garbage collection between each test to no avail.

A clue, hinting at issues related to the memory management, is the appearance of outliers primarily for path indexing. The discrimination tree requires no set operations and therefore does not allocate as many intermediate results as path indexing. By building a tree of the required set operations and evaluating it, the runtime may be forced to run a garbage collection before the test is finished.

We can also observe larger issues for the insertion test, seen in Figure 4.6a. At a size of 100, the path index takes almost ten times as long as we would expect. Similarly, the discrimination tree is also affected at a size of 700. The lower insertion time for a size of 5000 when compared to a size of 3000 is also unexpected as they are both run for the same number of times and should not be significantly affected by chance as all the tests were run with 50 different seeds.

Although we were not able to track down the exact reason for the outliers at the time of writing, we are confident that they are unrelated to the effectiveness of the term indices in general: our results coincide with those of previous studies [19, 14, 10] and tests with sizes close to those of the outliers show results as expected.

5 Conclusion

5.1 Related Work

Our work was inspired by the work of Stickel [19] who introduced path indexing as an alternative to discrimination trees. In their work, they also showed that path indexing is a promising competitor but that each technique has advantages and drawbacks. This observation was further examined by McCune [14]. By considering various optimisations for both techniques, they arrived at a more detailed conclusion. A study of these indices, in addition to abstraction trees and substitution trees was also conducted in [10].

Although we embedded first-order path indexing in the higher-order context of Isabelle and generalised the index to store arbitrary values instead of terms, the results we obtained generally agree with said literature. The consensus is that path indexing is superior to the variant of discrimination trees used in Isabelle for insertion, deletion and instances retrieval. On the other hand, discrimination trees dominate for variants and generalisations queries. Besides the insertion and deletion operations, which are slightly slower due to the combination with termtables, the results agree with our findings.

One significant difference to previous results are our findings for unifiable queries. Although their results are mixed, discrimination trees are found to be either comparable or superior. While we also found them to be comparable at small sizes, path indexing is significantly faster when 200 or more terms are indexed. This may be a result of the applicative style disrupting the performance of discrimination trees or the random generation of terms not accurately representing the terms of real applications and requires further investigation.

5.2 Summary and Future Work

We presented both path indexing and discrimination tree indexing and described the process of adapting the path index to Isabelle/ML, including the embedding of first-order terms in higher-order terms and the generalisation to indexing arbitrary values. In addition, we also considered optimisations to speed up the set operations, whose performance is crucial.

By combining path indexing with termtables (balanced 2-3 trees) and using integer identifiers in place of terms for lookup comparisons in the path index, we not only sped up all set operations but also supplemented the slow variants query of the path index with the exact lookup of the termtable. This is an opportunity to investigate other potential combinations of term indices. In addition, the use of a termtable, that distinguishes variables, enables the implementation of an exact duplicate detection, negating some of the caveats mentionend in Section 3.3.1.

Although current term indexing techniques in Isabelle/ML are able to deal with higher-order terms, they are optimised for the first-order case. Generalising term indices to handle higher-order terms correctly is, unfortunately, highly complicated as some queries may no longer be decidable [8]. Nevertheless, overapproximating term indices have been investigated and offer acceptable performance [15, 12]. Implementing such indices may prove to be challenging but also highly desirable.

While implementing path indexing, we took care to unify the interface with that of discrimination trees. This allows a user of either structure to simply swap the used term index for the other one. Moreover, this clears the way to easily add additional term indices such as substitution trees, which offer better performance for queries while compromising in sertion speed and memory requirements [10].

Lastly, we evaluated the performance of our implementation of path indexing in comparison to the existing implementation of discrimination trees in Isabelle/ML. While our evaluation largely reflects the results of the literature, they must be taken with a grain of salt as the generated terms are, after all, artificial and do not accurately represent real applications. Evaluating their performance on actual theories in Isabelle should be, thanks to the interface, relatively simple once proof methods are adapted to the minor changes in the interface.

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Reading, Mass: Addison-Wesley, 1995. 685 pp. ISBN: 978-0-201-53771-0.

[2] *Archive of Formal Proofs*. URL: https://www.isa-afp.org/ (visited on 04/14/2021).

[3] J. C. Blanchette, A. Popescu, D. Wand, and C. Weidenbach. "More spass with isabelle". In: *International Conference on Interactive Theorem Proving*. Springer, 2012, pp. 345–360.

[4] N. G. D. Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: (), p. 12.

[5] L. Bulwahn. "The new quickcheck for Isabelle". In: *International Conference on Certified Programs and Proofs*. Springer, 2012, pp. 92–108.

[6] S. Böhme and T. Nipkow. "Sledgehammer: judgement day". In: *International Joint Conference on Automated Reasoning*. Springer, 2010, pp. 107–121.

[7] K. Claessen and J. Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Acm sigplan notices* 46.4 (2011). Publisher: ACM New York, NY, USA, pp. 53–64.

[8] W. D. Goldfarb. "The undecidability of the second-order unification problem". In: *Theoretical Computer Science* 13.2 (1981). Publisher: Elsevier, pp. 225–230.

[9] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, and others. "seL4: Formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 207–220.

[10] D. Knuth. "Comparison of indexing techniques". In: *Term Indexing*. Ed. by P. Graf. Red. by J. G. Carbonell, J. Siekmann, G. Goos, J. Hartmanis, and J. Leeuwen. Vol. 1053. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 201–231. ISBN: 978-3-540-61040-3 978-3-540-49873-5. DOI: 10.1007/3-540-61040-5_16.

[11] X. Leroy. "A formally verified compiler back-end". In: *Journal of Automated Reasoning* 43.4 (2009). Publisher: Springer, pp. 363–446.

[12] T. Libal and A. Steen. "Towards a Substitution Tree Based Index for Higher-order Resolution Theorem Provers". In: (), p. 13.

[13] R. Loader. *Notes on simply typed lambda calculus*. University of Edinburgh, 1998.

[14] W. McCune. "Experiments with discrimination-tree indexing and path indexing for term retrieval". In: *Journal of Automated Reasoning* 9.2 (Oct. 1992), pp. 147–167. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/BF00245458.

[15] B. Pientka. "Higher-order term indexing using substitution trees". In: *ACM Transactions on Computational Logic* 11.1 (Oct. 2009), pp. 1–40. ISSN: 1529-3785, 1557-945X. DOI: 10.1145/1614431.1614437.

[16] *Poly/ML Home Page*. URL: https://polyml.org/ (visited on 04/05/2021).

[17] A. Riazanov and A. Voronkov. "Vampire". In: *International Conference on Automated Deduction*. Springer, 1999, pp. 292–296.

[18] S. Schulz. "System description: E 0.81". In: *International Joint Conference on Automated Reasoning*. Springer, 2004, pp. 223–228.

[19] M. E. Stickel. *The path-indexing method for indexing terms*. SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTER, 1989.

[20] *The PolyML structure*. URL: https://polyml.org/documentation/Reference/PolyMLStructure.html#pointerEq (visited on 04/05/2021).

[21] M. Twain. "Set-based indexing". In: *Term Indexing*. Ed. by P. Graf. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1995, pp. 51–126. ISBN: 978-3-540-49873-5. DOI: 10.1007/3-540-61040-5_14.

[22] M. Twain. "Set-based indexing". In: *Term Indexing*. Ed. by P. Graf. Red. by J. G. Carbonell, J. Siekmann, G. Goos, J. Hartmanis, and J. Leeuwen. Vol. 1053. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 51–126. ISBN: 978-3-540-61040-3 978-3-540-49873-5. DOI: 10.1007/3-540-61040-5_14.

[23] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. "SPASS Version 3.5". In: *International Conference on Automated Deduction*. Springer, 2009, pp. 140–145.

[24] M. Wenzel. *The Isabelle/Isar Reference Manual*. Feb. 20, 2021.