# Contents

# 1 Introduction

## 1.1 Motivation

## 1.2 Contributions

## 1.3 Thesis Outline

# 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 $\mathcal{V} \coloneqq \{x, y, z, x_1, x_2, ...\}$, all constants from the infinite set $\mathcal{C} \coloneqq \{a, b, c, c_1, c_2, ...\}$ and all functions from the infinite set $\mathcal{F} \coloneqq \{f, g, h, f_1, f_2, ...\}$. 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 $\mathcal{T} \coloneqq \{t, u, v, t_1, t_2, ...\}$, 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 $\rho$. We write $[t_1/x_1, ..., t_n/x_n]$ for the substitution $\{x_1 \mapsto t_1, ..., x_n \mapsto t_n\}$.

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 $\rho$, it is not modified in the term. Note that $\rho$ 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 $\rho$ such that $t\rho = u$. Similarly, we call $t$ and $u$ unifiable if and only if there exists a substitution $\rho$ such that $t\rho = u\rho$. In this case, $\rho$ 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 $\rho$ 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. [6]

### 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 $\lambda$-calculus is a formal language used to express computation based on functions. It is defined by a grammar for constructing $\lambda$-terms and rules for reducing them. The set of terms $\mathcal{T}$ of the untyped $\lambda$-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 [5].

## 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 $\lambda$-terms within which theorems and inference rules are embedded. [14]

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 $\lambda$-terms are a variant of simply typed $\lambda$-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 [2].

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 $\lambda$-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 $\lambda$-term $(\lambda x\ y.\ f\ x\ y)$.

We can embed first-order terms in these $\lambda$-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 `Bound`s, 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 [4], 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 [4]. 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 $f(x_1, ..., x_n)$ as internal nodes with the $x_i$ as children. Within this tree, every symbol has a position by the 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 in the term, which we call its path. The paths start at the top symbol and end with the index at which the symbol 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 a 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 $t$.

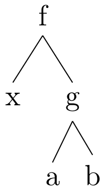
| Tree Representation | Path $p$ | $Symbol_t(p)$ |
|---|---|---|
|  | $\langle \rangle$ | $f$ |
| | $\langle (f, 1) \rangle$ | $*$ |
| | $\langle (f, 2) \rangle$ | $g$ |
| | $\langle (f, 2), (g, 1) \rangle$ | $a$ |
| | $\langle (f, 2), (g, 2) \rangle$ | $b$ |

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 the path defines the position of $symbol$ in the term. For example, the constraint $(\langle (f, 1) \rangle, c)$ is only fulfilled by terms of the form $f(c, ...)$. 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.

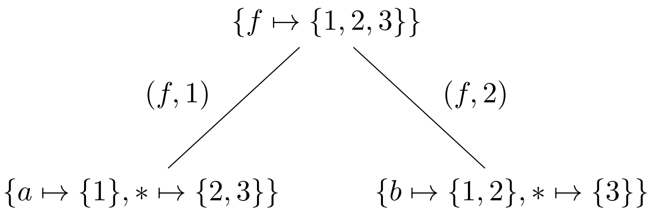
### 3.1.1 Structure

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.


$\{f \mapsto \{1,2,3\}\}$
$(f,1)$
$(f,2)$
$\{a \mapsto \{1\}, * \mapsto \{2,3\}\}$
$\{b \mapsto \{1,2\}, * \mapsto \{3\}\}$

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

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. Figure 3.3 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* returns *Terms*.

| Arguments<br>Query | $Q(p, x)$ | $Q(p, a)$ | $Q(p, f(t_1, ..., t_n))$ |
|---|---|---|---|
| $Q =$ variants | $Terms(p \cdot *)$ | $Terms(p \cdot a)$ | $\bigcap_i Q(p \cdot f \cdot i, t_i)$ |
| $Q =$ instances | $AllTerms$ | $Terms(p \cdot a)$ | $\bigcap_i Q(p \cdot f \cdot i, t_i)$ |
| $Q =$ generalisations | $Terms(p \cdot *)$ | $Terms(p \cdot a) \cup Terms(p \cdot *)$ | $\bigcap_i Q(p \cdot f \cdot i, t_i) \cup Terms(p \cdot *)$ |
| $Q =$ unifiables | $AllTerms$ | $Terms(p \cdot a) \cup Terms(p \cdot *)$ | $\bigcap_i Q(p \cdot f \cdot i, t_i) \cup Terms(p \cdot *)$ |

Figure 3.3: 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 $f(x_1, ..., x_n)$ is $\langle f, Preorder(x_1), ..., Preorder(x_n) \rangle$ 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 fig. 3.4. 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.

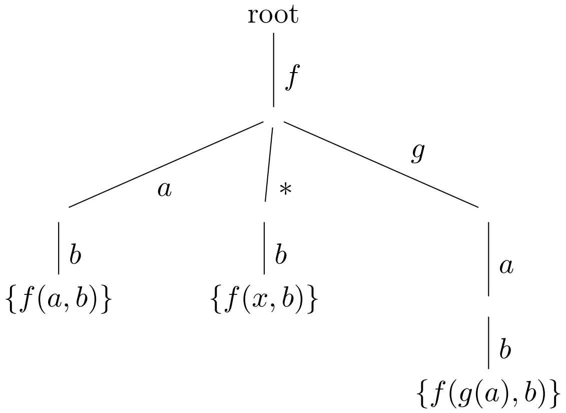


Figure 3.4: A discrimination tree index storing three terms

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 fig. 3.4 would traverse the tree by following the edges $\langle f, *, b\rangle$, retrieving the term $f(x,b)$

**Definition 3.4.** $slp(N,s)$ is the symbol lookup operation. It returns 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 $slp(slp(slp(N,a),b),c)$, as $slp(N,\langle a,b,c\rangle)$

**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 $terms(slp(root,\langle f,*,b\rangle))$. 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 $slp(N,c)$ as well as $slp(N,*)$. 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 $\mathit{terms}(\mathit{slp}(M,\langle a,b\rangle))$ and $\mathit{terms}(\mathit{slp}(M,\langle *,b\rangle))$ where $M=\mathit{slp}(\mathit{root},f)$, in addition to the empty sets of $\mathit{terms}(\mathit{slp}(\mathit{root},*))$ and $\mathit{terms}(\mathit{slp}(\mathit{root},\langle f,a,*\rangle))$.

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 fig. 3.4 are instances of $f(*,b)$. We notice that at the node $M=\mathit{slp}(\mathit{root},f)$, 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 $x_1,...,x_n$ of $g(x_1,...,x_n)$. When continuing in the $g$ branch, we must not continue at $\mathit{slp}(M,g)$, but at $\mathit{slp}(M,\langle g,a\rangle)$, as the variable is substituted by $g(a)$.

**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 $\text{arity}(c)=0$ we return $\mathit{slp}(N,c)$ (which is a direct child of $N$). For a function $f(x_1,...,x_n)$, we return the nodes $\mathit{skip}^n(\mathit{slp}(N,f))$. If for all $x_i$ $\mathit{arity}(x_i)=0$, $\mathit{skip}^n$ retrieve the nodes $1+n$ levels below $N$.

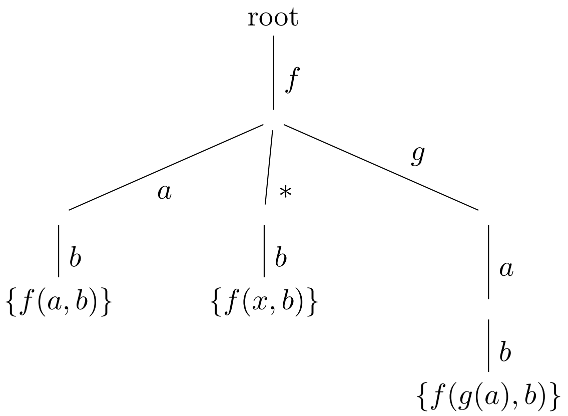


Figure 3.5: Skipping in the discrimination tree index

Draw arrows for skips

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 fig. 3.6. 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.

## 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` [10], a

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

Figure 3.6: The recursive definition of the queries

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 some attributes to, for example, differentiate between introductory and simplifying 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 values stored, 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 is supplied by the user and can, therefore, also be a constant $eq\ (v_1, v_2) = false$.

### 3.3.1 Caveats of current Implementation

The generalisation of storing terms to storing arbitrary 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.

Insertion and deletion uses key-value pairs with the $Preorder(t)$ being the key. This results in some potentially surprising behaviour. We illustrate this with some examples. We write $(t, v)$ for key-value pair 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 [11, 7, 13] 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 and performs well. Note that we need only check if one path, for example the top symbol, stores the same key-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 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 3.7 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 $f(g(c))$, due to sharing the $(p, Symbol_t(p))$

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 comparison of the values. As the comparison for values is supplied by the user, it may be arbitrarily slow, e.g. when storing large lists.

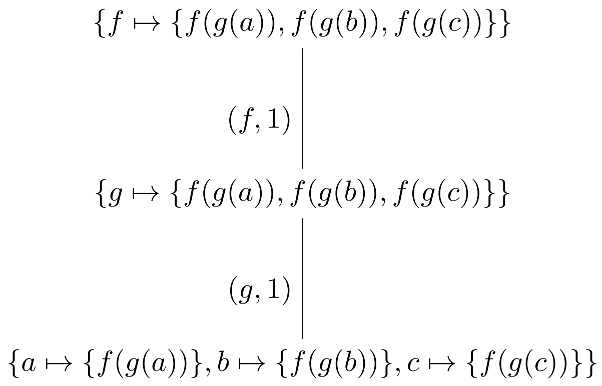

$\{f \mapsto \{f(g(a)), f(g(b)), f(g(c))\}\}$
$(f, 1)$
$\{g \mapsto \{f(g(a)), f(g(b)), f(g(c))\}\}$
$(g, 1)$
$\{a \mapsto \{f(g(a))\}, b \mapsto \{f(g(b))\}, c \mapsto \{f(g(c))\}\}$

Figure 3.7: A path index sharing all paths

We optimize this approach by using unique identifiers for each key-value pair. By instead storing an identifier, together with the value, in the path sets, we avoid repeatedly comparing $t$ with other terms. As the identifiers are unique for each key-value pair, we can also avoid repeatedly comparing $v$ with other values stored under the same term $t$. In essence, we use only the identifier to handle the path index operations correctly.

To associate each key-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 $(id, v)$, using $t$ as key. 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 gain 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 need only compare the identifier.

This approach provides us with an opportunity. By using the exact lookup of `termtable`, we can improve the duplicate detection to no longer ignore variable identity. In addition, we can also provide an exact lookup operation for the values by using only the table. 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. Nevertheless, 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 a single integer comparison. By using ordered lists, provided by `OrdList` in Isabelle/ML [8], 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. The fast integer ordering ensures that we can use `OrdList` at almost no overhead.

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 3.8 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 simplify the tree by removing *AllTerms* and the intersection, resulting only in $\{1\}$ remaining.

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

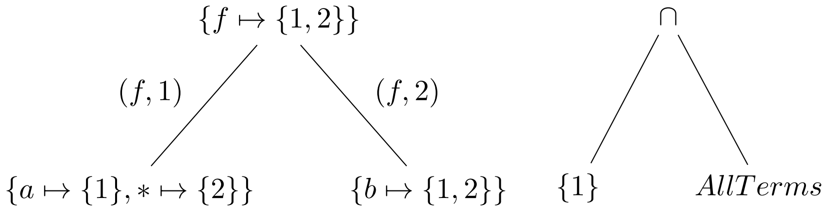


Figure 3.8: The operations tree for instances of $f(a,y)$

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, 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 the cache usage.

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 [9] which provides data sharing. This results in copies of immutable values requiring almost no additional memory. A related feature is `pointer_eq` [12] which is based on the data sharing and compares immutable values quickly. Unfortunately, we cannot directly take advantage of this as the user may wish to use a constant $eq(x, y) = false$ for comparison.

# 4 Testing in Isabelle/ML

## 4.1 Previous Work

A testing framework was built which tries to mirror QuickCheck from Haskell. Lack of typeclasses => Either use functors (OCaml, verbose), Compiler directly (Write tests as strings, no editor assistance, somewhat awkward) or explicitly pass generators, shows, shrinks etc.

Last option best because: Default generators can be provided but more complicated tests need custom generators anyway to satisfy preconditions etc., simple to use, simple to extend.

Modularity was relatively bad

## 4.2 Term Generation

Approaches: Random or deterministic Carry state around? Yes as we want maximum control. Possibly bad for performance as no multithreading this way. Use symbol generator to give maximum control over structure. Discarded idea: Separate generation into structure and symbols. Too intertwined to be efficiently separated. Address symbols by: Level + Index in Level or Path from root to symbol Deterministic generator with non-deterministic symbol generator works best => Useful for all cases but also simple to use. Symbol generator contains real complexity, term gen relatively basic (basically fold over yet to be generated tree)

## 4.3 Implementation Details

### 4.3.1 Overview of Modules

Shrinking: Generate simpler test cases from failed tests. Simplify repeatedly until no longer possible. Depth-First with only one level of Backtracking (or rather none and one consistency check before descending into child). Performance sensitive. Unfortunately the shrinking function must be provided. General shrinking is difficult as generator take no size argument. Else, we could simply take each involved generator and shrink their size (i.e. shrinkg listgen (itemgen 3) 10 returns [listgen (itemgen 2) 10, listgen (itemgen 3) 9]). Combinatorial explosion so not possible. Potential solution: Use compiler here as this is not exposed to the user if no shrink is given (by providing default shrinks for each type and applying them to the provided gen. Would require transparent generators where we can determine in retrospect which symbol gen was used for termgen etc.)

Output Style: Multiple output styles possible. No textfile output at the moment

Inputs: Lazy for performance, can take pregenerated lists of generators e.g. read from file. Constant values are also possible (e.g. ensure that previous failures do not fail again <- Not easily possible but with custom output should be doable: Append failed tests to file, rerun them for every test)

Lehman(?)-PRNG implemented, works as expected.

## 4.4 Usage

Generators take states. This state is often only a random variable but may contain other values. Generators can take other generators as argument and pass the state around. Deterministic gens dont require random values. Examples

## 4.5 Usage in this Thesis

What tests were written? What generators used?

# 5 Evaluation

Is PI better than DT? Theory and randomly tested

## 5.1 Expectations

Theoretical analysis, ref to comparison paper. In what cases is what approach superior? Prefix sharing

## 5.2 Approach

Randomly generated. Generation difficult, details on generator Focus on variable appearance as prefix sharing is even more difficult to reproduce

## 5.3 Comparison of PITT and DT

### 5.3.1 Variants

PI a bit better

### 5.3.2 Instances

PI significantly better

### 5.3.3 Generalisations

DT significantly better

### 5.3.4 Unifiables

PI significantly better

### 5.3.5 Maintenance Operations

PITT ca equal at insert, sig better at delete

### 5.3.6 Performance over Variable Frequency

At what point is break-even. Move perhaps as subsubsection into every query above.

### 5.3.7 Performance over Size

PITT scales better perhaps?

## 5.4 Comparison of PITT and PI

PITT significantly better

### 5.4.1 Insert

### 5.4.2 Delete

## 5.5 Recommendation

PI never PITT whenever: exact lookup, many vars, many unifs, many instances DN whenever, few vars, many gens

## 5.6 Shortcomings

Generators do not represent real applications Tests are noisy due to ?. PolyML?

# 6 Conclusion

## 6.1 Future Work

Test with real world application Test with different generators Benchmarks with less noise (larger size? Other environment?) Implement Substitution Trees, possibly better in every respect?

SpecCheck Improve Shrinks Consider anti-quotations more term gens

# 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] 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.

[3] *Discrimination Tree in Isabelle.*

[4] 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.

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

[6] 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.

[7] 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.

[8] *ord_ list.ML · isabelle.* URL: https://isabelle-dev.sketis.net/source/isabelle/browse/default/src/Pure/General/ord_list.ML;89cf7c903acad179e40c10f2f6643d3c2 (visited on 04/05/2021).

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

[10] *Termtables in Isabelle.*

[11] “The Path-Indexing Method for Indexing Terms”. In: (), p. 28.

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

[13] 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.

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