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 $V = \{x, y, z, x_1, x_2, ...\}$, all constants from the infinite set $C = \{a, b, c, d, c_1, c_2, ...\}$ and all functions from the infinite set $F = \{f, g, h, f_1, f_2, ...\}$. Whenever possible, we use only the first three symbols of each set for better readibility.

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

A term in first-order logic is a symbol s applied to $arity(s)$ arguments. We require all symbols s to always be applied to an $arity(s)$-tuple of terms.

2.1.1 Generalisation and Unification

A substitution is a partial function $\rho : V \to T$. We denote by $t\rho = u$ the term obtained by applying the substitution ρ to all variables v both in t and the domain of ρ. We write $[t_1/x_1, ..., t_n/x_n]$ for the substitution $\{x_1 \to t_1, ..., x_n \to t_n\}$. Note that ρ is applied only once to the original term, that is, $x[y/x, a/y] = y$ even though y would be substituted by a in the original term.

Given two terms t, u, we say that t is a generalisation of u and u a 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 and ρ a unifier if $t\rho = u\rho$. It can be seen that whenever t is a generalisation of u they are also unifiable.

2.2 Lambda Calculus

Define lambda calculus

2.3 Isabelle

Isabelle is a generic interactive theorem prover supporting formal theories. 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 the λ-terms within which the theorems and inference rules are embedded.

Isabelle is written for the most part in Standard ML (SML) and can also be extended at runtime. It is divided into a kernel, that has been verified and verifies the correctness of all proofs, and a userspace within which we can implement deduction methods for proofs.

2.3.1 Term Representation in Isabelle

The λ-terms in Isabelle/Pure are used in many different contexts and are also used for embedding first-order logic terms. They are a variant of simply typed λ-calculus and are built up recursively in an applicative style. They are defined, with minor differences 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. Their distinction is related to the metalogic nature of Isabelle/Pure and is irrelevant in the context of term indexing.
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. `Abs` is an abstraction whose variable is given a name for better readability.
5. $ represents the application of the first argument to the second one. As there are no tuples in this term representation, all functions are curried by default. $ is written infix.

We will ignore the types of terms and simply assume type correctness at all points. The λ-term $(\lambda x.\ x)\ a$ can then be represented directly as `Abs "x" (Bound 1)) $ Const "a"`. The application 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")`.

We can embed first-order terms in these λ-terms. While variables with an arity of 0 and constants map directly to `Var` and `Const` respectively. Likewise, the symbol of a function maps to a `Const`. The function as a whole, the symbol and the tuple of terms which are the arguments, is represented by a chain of applications with each component mapped to a λ-term. The term $f(a, g(x))$ is then represented by `Const "f" \$ Const "a" \$ (Const "g" \$ Var "x")`. Note the braces around $g(x)$ to differentiate this term from $f(a, g, x)$.

We assume every term to consist of only `Const`, `Var` and *textdollar*. This avoids the overhead of distinguishing `Const` and `Free`. `Abs` are not required for first-order terms and dangling `Bounds`, that is, indices pointing to a non-existing abstraction, are excluded, too.

3 Term Indexing

A term index efficiently stores terms while also providing efficient operations for the retrieval of variants[1], instances, generalisations and unifiables of a query term from the term index. Depending on the exact data structure, we can also implement other operations efficiently if desired, for example checking if a term is already stored, only inserting a term if no more general term is stored and efficiently merging two indices. In this thesis we limit ourselves to these operations as it is difficult to predict the exact use cases.

As generally presented in literature, term indices provide efficient, but overapproximating, queries. We disregard variable identities as it is prohibitively expensive to track the exact substitution while traversing the tree. By disregarding variable identities, we assume that each variable is unique and, as a result, avoid the occurs check. As such, a term index can only be used as a pre-filter whose results must, when necessary, be further filtered. We additionally disregard types as we expect the user of the index to only store consistent and type correct terms. Specifically, we assume that the arity of a function is fixed.

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

Instead of storing a term as a tree of functions and their arguments, we can specify the structure and symbols of a tree by combining every symbol of a term with its position, which we call its path.

Definition 3.1. The path is a sequence of $(symbol, index)$ pairs where the index describes the index of the next argument to traverse.[2]

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

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

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

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

A $(path, symbol)$ pair can be interpreted as a constraint on a term where the path defines the position of the symbol in the term. For example, $(< (f, 1) >, c)$ is only fulfilled by

[1] Identical up to variable identity

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

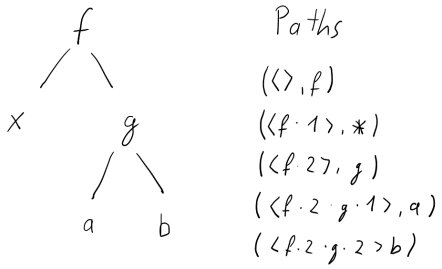

Figure 3.1: A term and its paths together with the symbols associated with them

terms of the form $f(c, ...)$[3]. A term gives raise to a set of $(path, symbol)$ pairs, which, when interpreted as constraints, uniquely identify this term up to loss of variable identitification.

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

Better phrasing

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

Drop "path set" completely?

Storing the path sets such that they can be quickly looked up by a $(path, symbol)$ pair can be achieved in multiple ways. We decided to use a trie-based approach as many of the paths share prefixes. The nodes of the trie contain a function $g : Symbol \longrightarrow 2^{Term}$. The edges are labelled with $(symbol, index)$ pairs, which correspond to the elements of a path. When we insert a path p of a term t we start at the root and traverse the trie according to p. Once we reach the end of p we extend g of the current node by $Symbol_t(p) \longrightarrow \{t\}$. To insert a term we simply insert all the paths that describe this term. This requires the insertion of many similar paths which profits from the prefix sharing.

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

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

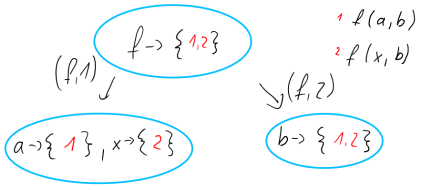

Figure 3.2: A path index

We are interested in retrieving the instances, generalisations and unifiables of a term stored in the index. In addition, we define a lookup to retrieve copies of the term. This

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

can be used to check if a term is already contained but may also be of interest as different variables are not distinguished. The queries are based on intersections and unions of the different path sets to enforce constraints on the terms.

To answer the simplest query, the lookup, we procede as follows:

1. Compute the set of $(path, symbol)$ pairs describing the term.

2. Retrieve the path sets corresponding to them in the index.

3. Intersect the path sets to retrieve all terms containing the same symbols at identical paths as the query term.

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

To retrieve the unifiables of a term from the index, we can use some observations regarding the unification problem.

1. A variable is unifiable with any other term

2. Constants are unifiable with themselves and variables

3. A function $f(x_1, ..., x_n)$ is unifiable with term t if and only if $t = x$ or $t = f(y_1, ..., y_n)$ where for all i x_i is unifiable with y_i. Again, a differing number of arguments are impossible as their types would clash otherwise.

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

PathTerms(p) refers to the path set stored at the path p. AllTerms is the collection of all terms stored in the index.

Arguments \ Query	Lookup	Instances	Generalizations	Unifiables
p x	PathTerms(p·*)	AllTerms	PathTerms(p·*)	AllTerms
p a	PathTerms(p·a)	PathTerms(p·a)	PathTerms(p·*) ∪ PathTerms(p·a)	PathTerms(p·*) ∪ PathTerms(p·a)
p $f(t_1, \dots, t_n)$	$\bigcap_n$ Lookup($p \cdot f \cdot n, t_n$)	$\bigcap_n$ Instances($p \cdot f \cdot n, t_n$)	PathTerms(p·*) ∪ $\bigcap_n$ Generalisations($p \cdot f \cdot n, t_n$)	PathTerms(p·*) ∪ $\bigcap_n$ Unifiables($p \cdot f \cdot n, t_n$)

Figure 3.3: The different queries and their definition

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 terms at its leaves and symbols at its internal nodes. To determine the leaf at which a term is stored we use the preorder traversal of the term which we obtain by simply reading the written term from left to right.

Definition 3.4. $Preorder(t)$ is the sequence of symbols obtained by the preorder traversal of the term t. For terminals it is the symbol itself. The preorder traversal of function $f(x_1, ..., x_n)$ is $< f, Preorder(x_1), ..., Preorder(x_n) >$. For the sake of simplicity, we flatten the sequence, i.e. $< f, < g, x >>$ becomes $< f, g, x >$.

For example, the preorder traversal of $t = f(c, g(x, y))$ is $< f, c, g, *, * >$[4].

We store the mapping $Preorder(t) \rightarrow t$ in a trie. Due to prefix sharing among the preorders of the different terms this reduces the memory consumption. The leafs contain the terms while the internal nodes only store the symbol by which they are addressed. A discrimination tree storing multiple terms can be seen in 3.4.

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

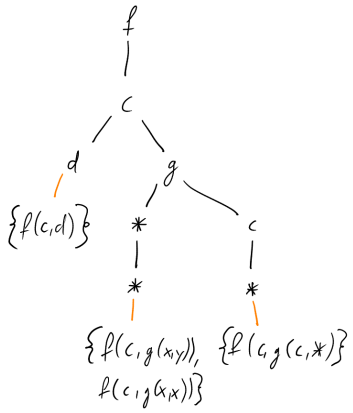

Figure 3.4: Multiple terms stored in a trie and indexed by the sequence obtained from their preorder traversal

Root =/= f, besser auf Beispiele ausgelegt: f(c,x) enthalten, netskip kommt vor etc.

The queries are implemented as a recursive algorithm on the nodes of the trie and $Preorder(t)$ 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. In some cases, we may also remove the arguments of the symbol. Therefore, we drop either the first symbol

[4]We ignore variable identity as it introduces significant complexity

s or $1 + arity(s)$ symbols from the sequence.

Definition 3.5. $slp(N, c)$ is the symbol lookup operation. It returns the child node of N representing the symbol c. If no such node exists, we return the empty node.

Retrieving the variants of a term is fairly straightforward. Starting at the root, we traverse the trie according to the preorder traversal using *slp*. By doing so, we ensure that we only retrieve terms that contain the same symbols (disregarding variable identity) in the same order, i.e. the variants of the term.

For example, we retrieve the variants of the term $t = f(c, x)$ in the discrimintation tree 3.4 in the following manner, written as $(Node, Preorder(t))$: $(root, < f, c, x > \rightarrow (f, < c, x >) \rightarrow (c, < x >) \rightarrow (*, <>)$ At this point we have reached a leaf containing the term $f(c, x)$, the only stored variant of the term.

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

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

Definition 3.6. $skip(N)$ returns the set of nodes obtained by skipping a single term starting at N. For all symbols s for which $slp(N, s)$ is defined, we collect all nodes $1 + arity(s)$ levels below N. That is, for a constant c with $arity(c) = 0$ we return $slp(N, c)$ (which is a direct child of N). For a unary function f we return the nodes $slp(slp(N, f), x)$ of all symbols s.

Using this, we can retrieve all the nodes reached by replacing the variable in the query term with some term. The union of the terms returned by the query on each node represents the result. An overview of all the queries is given in 3.5. 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.

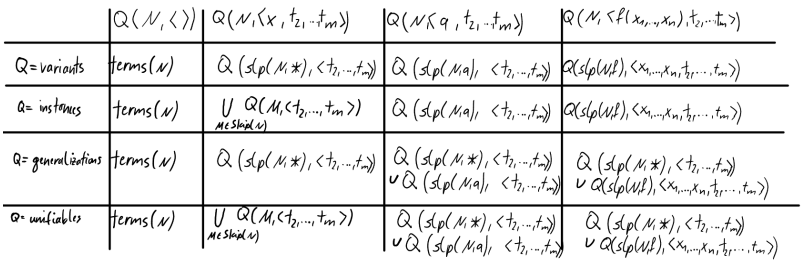

Figure 3.5: The definition of the different queries

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`, 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, `termtables` do not offer any support for the more complex queries.

3.3.1 Discrimination Trees

The need for efficient retrieval of unifiables and generalisations of a query term is addressed by a discrimination tree implementation. Despite being based on the concept introduced above, the discrimination tree implementation in Isabelle/ML stores arbitrary sets of values indexed by terms. This allows us to, for example, store horn clauses for efficient backward chaining. By storing the premises of a clause in the node addressed by its conclusion, we can later query our knowledge base for unifiables of our current goal. On successful retrieval, we can replace the goal with the set of premises.

```
datatype 'a T =
  Leaf of 'a list
| Net of {atoms: 'a T Symboltable.T, comb: 'a T, var: 'a T}
val insert: ('a * 'a -> bool) -> term * 'a -> 'a T -> 'a T
val content: 'a T -> 'a list
val delete: ('a -> bool) -> term -> 'a T -> 'a T

val lookup: 'a T -> term -> 'a list
val instances: 'a T -> term -> 'a list
val generalisations: 'a T -> term -> 'a list
val unifiables: 'a T -> term -> 'a list
```

Figure 3.6: A subset of the signature of the discrmination tree

As we can see from the signature in 3.3.1, the discrimination tree stores a list of arbitrary values in each leaf node. The internal nodes, named `Net`, store the children separated into three separate trees. `var` stores both variables and abstractions and atoms stores the constant symbols. Due to the applicative nature of the terms, we also need a

continue here

The generalisation of storing terms to storing arbitrary single values is relatively simple for discrimination trees but, unfortunately, significantly more difficult for path indexing. This compounds with the fact, that the discrimination tree implementation does not only store arbitrary values but instead efficiently stores sets of arbitrary values. While this does not complicate the datastructre (after all, a data structure for arbitrary values can also store sets of values), the semantics of `insert`, `delete` and duplicate detection are consequently more complicated.

We illustrate this with some examples. We use $(term, value)$ for the items stored and DT for the (initially) empty discrimination tree. We use the syntactic equality although we can avoid deduplication by using a non-reflexive comparsion function:

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. Inserting (x, x) and (y, y) into DT stores both variables x and y in the same node as, by α-equivalence, the values are different.[5]

4. 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. Unfortunately, we can delete this value not only with the term x but also with y and $\lambda x.x$ as these are considered to be equivalent by the index. Additionally, deleting a value completely, that is, from all locations it is stored at, is not possible.

The above may not seem too surprising and can be seen as a natural consequence of the design constraints. For example, storing sets of values nicely sidesteps the problem of distinguishing higher-order terms. The discrimination tree can handle higher-order terms without any problems by mapping all λ-expressions to variables. Therefore, it will return, perhaps extreme, overapproximations to queries but will not fail to store the values or erroneously detect duplication.

Mention mapping of HOL to FOL where?

The lack of deduplication in queries is a consequence of ease-of-use and consistency. As the insertion of the identical value at different nodes succeeds the different instances of the value should be treated differently. Furthermore, storing the same value multiple times is most likely a rare occurence in which the user is responsible for correct handling.

Nevertheless, this represents significant complexity which, while natural for discrimination trees, must be carefully reproduced in the path index implementation. We recall that a term is never explicitly stored in path indexing as we represent a term by a collection of paths which each store a reference to the term. Naively replacing the reference to the term by a reference to a set of values changes the semantics significantly. Different terms often share at least one path but by no longer storing a reference to the term we lose the information from which term a $(path, value)$ pair originates. Implementing the deletion correctly is no longer possible.

Continue here

Unfortunately most literature on path indexing only covers the storage of terms. The queries of path indexing rely on the intersection and union of the path sets. These in turn rely on the fast comparison of the stored values. For example, storing hash tables in the path index would be extremely slow as they cannot be compared directly[6] and comparing the contents requires the collection of all entries. To solve this potential problem we investigated multiple solutions.

2 Quellen sicher, noch welche?

Furthermore, the insertion of an identical $(term, value)$ pair raises an exception. We require a comparison function during insertion to determine this.[7] As the index only stores the terms in the path sets we have to store $(term, value)$ pairs in the path sets. Assume

[5] The term equality used by the discrimination tree to map the higher-order terms to the internal first-order terms without variable identity is not exposed.

[6] The structure of the hashtable depends on the insertion order

[7] The comparison function need not be reflexive, for example the constant (λx.false) is valid.

we store only the values in the path sets. We cannot determine whether, for example, the path index containing $(f(c), 1)$ and $(g(d), 1)$ has also stored the $(f(d), 1)$ as the value 1 is present in all path sets associated with $f(d)$, namely $(<>, f)$ and $(< f, 1 >, d)$.

The first approach requires a comparison function for the values.[8] Using the index becomes more difficult by doing so. A user has to implement a comparison function for values and additionally has to consider the potential performance impact. This can be partially mitigated by using pointerEq, although it can only be used as a shortcut for identical values. The comparison must still be called for differing values since there is no perfect sharing.

Code references

Remove paragraph? Doesn't work with (term,value) pairs

The second approach is the storage of $(term, value)$ pairs. By doing so we can implement all the operations according to the literature and simply discard the term before returning the results. This simplifies implementation and retains acceptable performance as the comparison of differing terms will likely only need to compare the first few symbols. It will also increase the memory consumption as a copy of every term is stored solely for the set operations. (Additionally there is no immutable pointer implementation in Isabelle/ML. Instead, copies of identical values are shared by the runtime.)

PolyML erwähnen?

Überhaupt nicht erwähnen weil Data Sharing gut genug funktioniert?

This approach can be further optimised by replacing the $(term, value)$ pairs by $(identifier, value)$ pairs and mapping each term to an identifier. By using integers as identifier, we reduce the comparison to an integer comparison. Additionally, we can use ordered lists, provided by the SML standard library, for the path sets to implement the set operations more efficiently. We are also less reliant on the pointer equality provided by Poly/ML and runtime details like the merging of identical immutable values. This is quite important as we do not have any guarantee when the last heap compression occurred and manual invocation by using the shareCommonData introduces significant overhead to insertion. Additionally, reliance on low-level functions like shareCommonData and pointerEq should be avoided as there are may be significant changes across runtime versions.

Reference

We can further speed up the set operations by building a tree of the intersections and unions and only evaluating it at the end. This likely utilizes the cache better because the previously calculated list is not evicted from the cache by the trie traversal. Furthermore, this presumably enables further compiler optimizations as the intermediate results are only short-lived and functions can be inlined.

Example, too vague

Data Sharing in Poly/ML. NoConstraint exc. No generic hash. Saving "Copy" of values because pointers/ref are always mutable and bad for GC etc.

3.3.2 Delete Semantics

There already exists an implementation of a term index in Isabelle/ML with a two caveats. Firstly,

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

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

5.1 Testweise

5.2 Vergleich von Queries, mit DN, Itemnets

5.3 Vergleich von Insert, Delete, (Merge); mit DN, Itemnets, Termtabs

5.4 Wann PI hernehmen statt dem Rest?

5.5 Shortcomings

5.6 Future Work

6 Conclusion

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Reading, Mass: Addison-Wesley, 1995. 22–25. ISBN: 978-0-201-53771-0.