

LUDWIG-MAXIMILIANS-UNIVERSITY OF MUNICH
TEACHING AND RESEARCH UNIT PROGRAMMING AND MODELLING
LANGUAGES

PROF. DR. FRANÇOIS BRY

Implementation and Comparison of Substitution Trees and Instance Tries

Bachelor Thesis

in the course of studies 'Computer Science plus Mathematics'

presented by

Lydia Kondylidou

Supervisors: Prof. Dr. François Bry, Thomas Prokosch

Deadline: March 26, 2021

Lydia Kondylidou

Offenbachstrasse 41, 81245

Munich, Germany

L.Kondylidou@campus.lmu.de


Matrikelnr.: 11055585

Declaration

I hereby assure that this project is the result of my own work, except where explicit reference is made to the work of others.

I also confirm that this work has not been submitted for another qualification to this or any other university.

Munich, the March 26, 2021



Acknowledgments

I would like to express my sincere gratitude to my supervisor, Thomas Prokosch. Throughout my thesis, he provided me with valuable guidance and help. He always found time for me in the middle of his busy schedule to go over the details and provide suggestions and direction.

I would like to thank the department Chairperson, Prof. Dr. François Bry, for giving me the opportunity to do my thesis and his review and comments.

Finally, I would like to thank my entire family for their feedback, support and love.

Abstract

Term indexing methods have a major impact on the speed of logic programming languages and automated theorem proving systems. Substitution trees are well established term indexes for logic programming and theorem proving because of their performances. However, they rely on heuristics and ignore the partial order that is inherent to terms. On the other hand, instance tries have been recently proposed as a novel term index for logic programming and automated theorem proving. Instance tries make use of the term order without having to resort to heuristics. In this bachelor's thesis, the efficiency of instance tries is examined in more detail on the basis of established benchmarks and it is compared with that of substitution trees.

Contents

1	Introduction	3
1.1	Motivation and overview	3
1.1.1	Motivation	3
1.1.2	Overview	4
1.2	The programming language Rust	4
2	Overview on term indexing	6
2.1	Automated reasoning	6
2.2	Automated theorem proving	6
2.3	Logic Programming	6
2.4	Term indexing	7
3	Substitution trees	8
3.1	Overview	8
3.1.1	Substitution	8
3.1.2	Idempotent substitution	9
3.1.3	Matcher	9
3.1.4	Positions in a term	9
3.1.5	Normalisation	9
3.2	Substitution trees and their implementation	10
3.2.1	Variant nodes	11
3.3	MSCG (Most Specific Common Generalisation)	11
3.3.1	Computing a common generalisation of two substitutions	12
3.3.2	Usage	12
4	Instance tries	14
4.1	Unification algorithms	14
4.1.1	Unifier	15
4.1.2	Instance	15
4.1.3	Matching and instantiation	16
4.2	Instance tries and their implementation	16
5	Performance and measurements	18
5.1	COMPIT (COMParing Indexing Techniques for automated deduction)	18

5.2 Test cases	19
5.3 Implementation	19
5.4 Runtime measurements	20
6 Code and data	20
7 Analysis	21
8 Conclusion	27
9 Future work and suggestions	30
Literature	31

1 Introduction

1.1 Motivation and overview

1.1.1 Motivation

First-order terms constitute the basic representational unit of information in several disciplines of computer science such as automated deduction, term rewriting, symbolic computing, and logic and functional programming. In contexts where the data sets get large and/or keep growing, as in automated theorem provers and deductive databases, new techniques are needed to speed up the operations on terms. The time saved from operations like retrievals of candidate terms, could be used for performing other useful computations.

The interest in term indexing has grown, as experiments pointed out that theorem provers that use generative procedures like resolution (Robinson,1965b; Chang and Lee, 1973) or Knuth-Bendix completion (Knuth and Bendix,1970) face the problem of performance degradation, as mentioned by Wos [1992]:

”After a few CPU minutes of use, a reasoning program typically makes deductions at less than 1 percent of its ability at the beginning of a run.”

Term indexing particularly influences a system’s performance by providing rapid access to first-order terms with specific properties. Several term indexing methods have been developed and several experiments have been conducted on them. Until now, substitution trees achieve maximal search speed paired with minimal memory requirements in various experiments and outperform traditional techniques such as path indexing, discrimination tree indexing, and abstraction trees by combining their advantages and adding some new features.

Advances in term indexing had great impact on the design and performance of automated reasoning systems in the last decade. As efficiency and progress remain the main focus of researchers, a new term indexing technique has been implemented, namely instance tries. The focus of this thesis is to validate experimentally that instance tries perform updates and retrieval operations faster than substitution trees. To evaluate both instance tries and substitution trees, firstly, both data structures have been implemented in the Rust programming language. Afterwards, the performance of both has been examined on the basis of established benchmarks. Lastly, the results from the instance tries and substitution trees have been compared and a conclusion

has been drawn.

1.1.2 Overview

This thesis is structured as follows. Chapter 2 gives a brief introduction on term indexing. Chapter 3 is dedicated to substitution trees and one of its central algorithm, the most specific common generalisation. In chapter 4, instance tries are being presented, as well as their key algorithm, the instance trie unification algorithm. At the end of chapter 4 the approach of the implementation of the instance tries is presented. In chapter 5, the focus lies on the test cases, as well as the performance and measurements. The results are being presented in Chapter 6 to 8 and in chapter 9 future work and suggestions are considered.

1.2 The programming language Rust

In term indexing systems speed is of utmost importance. The choice of a programming language dictates the speed of an application program. One programming language, which promises efficient code, is Rust.

Rust is a programming language, the runtime system of which is open source, that focuses on speed, memory safety and parallelism. Developers use Rust for a wide range of applications: Game engines,¹ operating systems,² data systems and browser components (Mozilla 2016). An active community of volunteer coders maintain the Rust programming language and continues to add new enhancements. Mozilla sponsored the Rust open source project (<https://www.rust-lang.org>) for several years.

Rust makes systems programming accessible by combining speed with efficiency. Using Rust, programmers can make software less prone to bugs and security exploits. Many statically-typed languages allow for the concept of NULL. Like Haskell and some other modern programming languages, Rust encodes this possibility using the type system.

One of the biggest benefits of using a systems programming language like Rust is the ability to have control over low-level aspects. Rust, in particular, offers the choice of storing data on the stack or heap and determines at compile time when memory is no longer needed and can be cleaned up. This allows an efficient use of memory as well

¹<https://arewefgameyet.rs>

²for example Redox OS

as a performant memory access.

Unlike many existing systems programming languages, Rust strives to have as many zero-cost abstractions as possible. To put it bluntly, the programmers' coding style should not affect the performance.

The biggest benefit Rust can provide compared to other languages is a borrow checker. This part of the compiler is responsible for ensuring that references, values that refer to certain data but do not own it, do not outlive this data. This provides the opportunity to eliminate entire classes of bugs caused by memory unsafety.

This benefit, can also become a difficulty of the Rust programming language. The following example creates a mutable string which contains a name and then applies a reference to the first three bytes of the name. While that reference is outstanding, we attempt to mutate the string by clearing it. Since there is no guarantee that the reference points to valid data and the action of dereferencing could lead to undefined behaviour, the compiler returns an error message:

```
fn no_mutable_aliasing() {
    let mut name = String::from("Vivian");
    let nickname = &name[..3];
    name.clear(); //Truncates this String, removing all contents.
    println!("Hello_ there ,_{}!", nickname);
}
```

```
error[E0502]: cannot borrow 'name' as mutable because it is also
borrowed as immutable
```

```
--> a.rs:4:5
   |
3 |     let nickname = &name[..3];
   |                               ---- immutable borrow occurs here
4 |     name.clear();
   |     ^^^^^^^^^^^^^^^^^ mutable borrow occurs here
5 |     println!("Hello_ there ,_{}!", nickname);
   |                                               ----- immutable borrow
   |                                               later used here
```

For more information about this error, try 'rustc --explain E0502'.

Helpfully, the error message incorporates the code and tries hard to explain the problem, pointing out exact locations. There is a reference pointing into the string, that the program has to clear. Doing so might require that the memory of the string be

freed, invalidating any existing references. In case this happens and it is used in the value of "nickname" we would be accessing uninitialised memory, potentially causing a crash.

2 Overview on term indexing

Term indexing supports the construction of efficient automated reasoning systems, as e.g. automated theorem provers, by allowing rapid access to first order predicate calculus terms with specific properties. Advances in term indexing had great impact on the design and on the performance of automated reasoning systems in the last decade. This chapter provides an overview on term indexing.

2.1 Automated reasoning

Automated reasoning is a sub-field of artificial intelligence (Graf 1996). The study of automated reasoning helps produce computer programs that assist in solving mathematical and logical problems. Starting with the formal description of a specific problem, an automated reasoning system draws conclusions that logically follow from the supplied facts (Bentley 1995).

2.2 Automated theorem proving

One of the most developed subareas of automated reasoning is automated theorem proving. Researchers in automated theorem proving seek to develop software that does non-trivial mathematics. More specifically, automated theorem proving is dealing with proving mathematical theorems by computer programs. At the most ambitious level, the hope is to eventually produce programs that can solve extremely difficult open problems (Graf 1996). Automated reasoning over mathematical proofs was a major impetus for the development of computer science.

2.3 Logic Programming

Another subfield of automated reasoning is logic programming. The aim of logic programming is to use formal logics as a programming language. Any program written in a logic programming language is a set of sentences in logical form, expressing facts

and rules about some problem domain. Major logic programming language families include Prolog, answer set programming (ASP) and Datalog.

2.4 Term indexing

Indexes are used in computer science to ensure fast access to data in a large data collection. Data is typically managed sequentially on a storage medium. Processing a search query without indexing would involve linear effort, in the worst case the entire data would have to be searched.

In general, an index can be seen as a means for accessing and retrieving data associated with a certain query key t from an index of standard database systems, by comparing the indexed keys s_i saved in the database with the query key t (Graf 1996).

In automated reasoning systems an index is used to access and retrieve data elements based on a query term t . For this, the indexed term denoted as s_i , needs to be compared with the query term t . The comparison of this retrieval operation will not only be based only on equality relations, but on several binary relations in automated theorem proving with the property $R(s_i, t)$. These relations are called retrieval conditions (Graf 1996).

Hence, the problem of term indexing can be formulated abstractly as follows. Given a set L of indexed terms, a retrieval condition R over terms and a query term t , identify the subset S of L that consists of the terms s_i such that $R(s_i, t)$ holds.

Most interesting retrieval conditions are formulated as existence of a substitution that relates in a special way the query and the retrieved objects s_i . Four of these retrieval operations of concern are variants, generalisations, instances, and terms that unify with a given term and are defined as follows.

Definition 2.1 (Standard Relations).

- **Unification:** The relation $Unif(s_i, t)$ holds if and only if s_i and t are unifiable, that is there exists a substitution σ such that $s_i\sigma = t\sigma$.
- **Instance:** The relation $Inst(s_i, t)$ holds if and only if s_i is an instance of t , that is there exists a substitution σ such that $s_i = t\sigma$.
- **Generalisation:** The relation $Gen(s_i, t)$ holds if and only if s_i is a generalisation of t , that is there exists a substitution σ such that $s_i\sigma = t$.

- **Variante:** The relation $Vari(s_i, t)$ holds if and only if s_i is a variant of t , that is there exists a renaming substitution σ such that $s_i\sigma = t$.

Such a retrieval of candidate terms in theorem proving is interleaved with insertion of terms to L , and deletions from L .

In order to support rapid retrieval of candidate terms, we need to process the indexed set into a data structure called the *index*. Indexing data structures are well-known to be crucial for the efficiency of the current state-of-the-art theorem provers (Nieuwenhuis et al. 2001).

Based on the observation that the performance of automated reasoning systems can be increased by using an index for the retrieval and maintenance of data, the main task of this thesis can be formulated as follows:

Present an existing term indexing technique, substitution trees, and a new term indexing data structure, instance tries, and compare them with each other on hand of experimental results.

3 Substitution trees

3.1 Overview

This section gives an introduction to the substitution tree indexing and defines substitution, idempotent substitution and normalisation of substitutions. Furthermore, it explains substitution trees and their implementation in this bachelors' project.

Let V and F be two disjoint sets of symbols. V denotes the set of variable symbols and $V^* \subset V$ is the set of indicator variables. The set of n -ary function symbols is F_n and $F = \cup F_i$, for $0 \leq i \leq n$. Furthermore, T is the set of terms with $V \subseteq T$ and $f(t_1, \dots, t_n) \in T$ if $f \in F_n$ and $t_i \in T$, for $0 \leq i \leq n$. The variables occurring in a term t or a set of terms are denoted by $V(t)$. Function symbols with arity 0 are called constants. In the following examples the symbols $u, v, x, y, z \in V, *i \in V^*, f \in F \setminus F_0$ and $a, b \in F_0$ are used.

3.1.1 Substitution

Example 1. A substitution σ is a mapping from variables to terms represented by the set of assignments $\{x_1 \rightarrow t_1, \dots, x_n \rightarrow t_n\}$. The set $DOM(\sigma) = \{x \in V \mid \sigma(x) \neq x\}$

is called domain of σ , the set $COD(\sigma) = \{\sigma(x) \mid x \in DOM(\sigma)\}$ the codomain of σ , and $I(\sigma) = V(COD(\sigma))$ is the set of variables introduced by σ .

3.1.2 Idempotent substitution

A substitution σ is called **idempotent** if and only if $\sigma\sigma = \sigma$, and hence $t\sigma\sigma = t\sigma$ for every term t . The substitution $\{x_1 \rightarrow t_1, \dots, x_n \rightarrow t_n\}$ is idempotent if and only if none of the variables x_i occurs in any t_i . Substitution composition is not commutative, that is, $\sigma\tau$ may be different from $\tau\sigma$, even if σ and τ are idempotent.

Example 2. The substitution $\{x \rightarrow y + y\}$ is idempotent, e.g. $((x + y)\{x \rightarrow y + y\})\{x \rightarrow y + y\} = ((y + y) + y)\{x \rightarrow y + y\} = (y + y) + y$, while the substitution $\{x \rightarrow x + y\}$ is non-idempotent, e.g. $((x + y)\{x \rightarrow x + y\})\{x \rightarrow x + y\} = ((x + y) + y)\{x \rightarrow x + y\} = ((x + y) + y) + y$.

3.1.3 Matcher

A substitution σ is called a **matcher** from term s to term t if $s\sigma = t$. In this case s is called a generalisation of t and t is called an instance of s .

3.1.4 Positions in a term

In this subsection the position in a term is defined ([Graf 1996](#)). Let $f(t_1, \dots, t_n)$, $n \in \mathbb{N}$, be a term t with subterms t_i , $1 \leq i \leq n$. The subterm t_i of the term $f(t_1, \dots, t_n)$ at the position p , is denoted as t/p and can be recursively defined as $t/p = t_i$ if the finite sequence of natural numbers p starts with i .

Example 3. Let a be a constant, $g(a)$ and $f(g(g(a)), g(a))$ be terms. The term $g(a)$ occurs at positions $[1, 1]$ and $[2]$ in $f(g(g(a)), g(a))$.

3.1.5 Normalisation

In this section normalisation of terms and substitutions is introduced. Normalisation is a variable renaming from the set of non-indicator variables $V \setminus V^*$ to the set of indicator variables V^* . The goal of normalisation is to allow for variable sharing between all substitutions stored in the index, thereby reducing memory consumption.

Normalisation of terms Let $s = f(t_1, \dots, t_n)$ be a term with p_1, \dots, p_m denoting positions of all unique variables in s . The substitution $\sigma = \{s/p_1 \rightarrow *i, \dots, s/p_m \rightarrow *m\}$ with $*i \in V^*$ is called normalisation. The normalised term for a term s is denoted by \bar{s} and it holds that $\bar{s} = s\sigma$, where σ is a renaming substitution.

Normalisation of substitutions Let $\sigma = \{x_1 \rightarrow t_1, \dots, x_n \rightarrow t_n\}$ be a substitution and $<$ an ordering on variables such that $x_1 < \dots < x_n$. The normalisation of σ , denoted by $\bar{\sigma}$, is based on the normalisation of terms. Therefore, let $t = f_n(t_1, \dots, t_n)$. The normalisation of t is $\bar{t} = \overline{f_n(t_1, \dots, t_n)}$. Now, the normalisation of σ is defined as $\bar{\sigma} = \{x_1 \rightarrow \bar{t}/1, \dots, x_n \rightarrow \bar{t}/n\}$

Example 4. Let f be a binary function symbol, σ a substitution, u, v, x, y variables, $*1, *2$ indicator variables and a a constant. If $\sigma = \{x \rightarrow f(u, v), y \rightarrow f(a, v)\}$ and $x < y$ then $\bar{\sigma} = \{x \rightarrow f(*1, *2), y \rightarrow f(a, *2)\}$. This is the case as the terms $f(u, v)$ and $f(a, v)$ are normalised to $f(*1, *2)$ and $f(a, *2)$, based on the above mentioned normalisation of terms. However, if we had chosen $y < x$ then σ would have the normalisation $\bar{\sigma}' = \{x \rightarrow f(*2, *1), y \rightarrow f(a, *1)\}$.

3.2 Substitution trees and their implementation

In this chapter substitution tree indexing is presented as an indexing technique. Substitution tree indexing (Graf 1996) is a highly successful first-order term indexing strategy which allows the sharing of common sub-expressions via substitutions. The labels of substitution tree nodes are substitutions (see chapter 3.1.1). Each path in the tree therefore represents a binding chain for variables. Consequently, the substitutions of a path from the root node down to a particular node can be composed and yield an instance of the root node's substitution.

Substitution trees can represent any set of idempotent substitutions 3.1.2. However, only paths from the root node to leaf nodes actually represent terms to be stored while shorter paths do not. This is in contrast to instance tries. In the simplest case all these substitutions have identical domains and consist of a single assignment, which implies that the substitution tree can be used as a term index as well (Graf 1996).

Example 5. We illustrate substitution tree indexing with an example set consisting of the following substitutions:

$$\{u \rightarrow f(a, b)\}, \quad \{u \rightarrow f(y, b)\}, \quad \{u \rightarrow f(b, z)\}$$

which obviously represent a term index for the terms:

$$f(a, b), \quad f(y, b), \quad f(b, z)$$

While substitutions are inserted into the index, their codomain is renamed. Normalisation [3.1.5](#) changes all variables in the codomain of a substitution. The substitutions inserted to the index in figure [1](#) therefore is $\{u \rightarrow f(a, b)\}$, $\{u \rightarrow f(*1, b)\}$ and $\{u \rightarrow f(b, *1)\}$. The following tree was produced by inserting each of these substitution into the index.

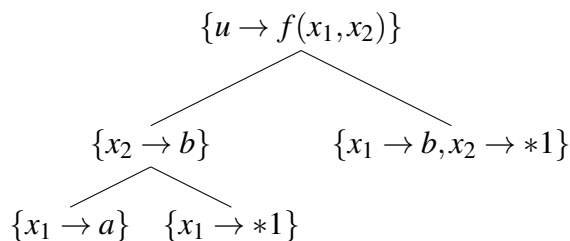


Figure 1: Substitution Tree

3.2.1 Variant nodes

In this section, the definition of variant nodes in substitution trees is presented. Variant nodes are needed in the update and retrieval operations of substitution trees. For example, when inserting an expression to a substitution tree, the position has to be first detected where the expression will be inserted. The search for variant nodes is going to determine this position.

Let $N = (\Sigma, \Omega)$ be a substitution tree and ρ a substitution. Then

$$V(N, \rho) := \{\sigma \mid \forall x_i \in \text{DOM}(\Sigma). \sigma(\rho(\Sigma(x_i))) = \rho(x_i) \wedge \text{DOM}(\sigma) \cap V^* = \emptyset\}$$

3.3 MSCG (Most Specific Common Generalisation)

To build a substitution tree, the MSCG (Most Specific Common Generalisation) between two expressions is needed. This operation is necessary to build and maintain substitution trees or retrieve elements from the index.

This section considers the computation and usage of the most specific common generalisation algorithm as presented in the book "Term Indexing" ([Graf 1996](#):154). Furthermore, it details its implementation of the aforementioned and finally, it presents the algorithms' performance, concluded by certain runtime and storage measurements and presents related work.

3.3.1 Computing a common generalisation of two substitutions

Let τ_1 and τ_2 be two substitutions. If substitutions μ , σ_1 and σ_2 exist such that $\mu \circ \sigma_1 = \tau_1$ and $\mu \circ \sigma_2 = \tau_2$, then μ is called a *common generalisation* of τ_1 and τ_2 . A common generalisation μ is called *most specific* (mscg) if there is a substitution λ for all common generalisations ν such that $\nu = \mu \circ \lambda$. The most specific common generalisation for τ_1 and τ_2 is computed by the function

$$mscg(\tau_1, \tau_2) := (\mu, \sigma_1, \sigma_2)$$

The substitution μ is called the *most specific common generalisation* (mscg) for τ_1 and τ_2 . The substitutions σ_1 and σ_2 are called *specialisations*.

Example 6. Suppose $\tau = \{x \rightarrow g(b), y \rightarrow a\}$ and $\rho = \{x \rightarrow g(a), y \rightarrow b\}$. Although $\{x \rightarrow g(x_1)\}$ and $\{x \rightarrow x_1\}$ are common generalisations of τ and ρ , we have

$$mscg(\tau, \rho) = (\{x \rightarrow g(x_1)\}, \{x_1 \rightarrow b, y \rightarrow a\}, \{x_1 \rightarrow a, y \rightarrow b\}).$$

The original substitutions τ and ρ can be reconstructed by $\tau = \{x \rightarrow g(x_1), y \rightarrow x_2\} \circ \{x_1 \rightarrow b, x_2 \rightarrow a\}$ and $\rho = \{x \rightarrow g(x_1), y \rightarrow x_2\} \circ \{x_1 \rightarrow a, x_2 \rightarrow b\}$, respectively. Note that x_1 and x_2 are new auxiliary variables. These auxiliary variables represent the parts of the substitutions which differ from each other.

There is more than one way to compute the MSCG, for example

$$mscg(\tau, \rho) = (\{x \rightarrow g(x_1), y \rightarrow x_2\}, \{x_1 \rightarrow b, x_2 \rightarrow a\}, \{x_1 \rightarrow a, x_2 \rightarrow b\}).$$

However, the computation of the most specific common generalisation for the use in substitution trees require the one mentioned above.

3.3.2 Usage

The aim of the implementation of the MSCG algorithm, is its use in one of the most important tree operations, the insertion of a tree element. The insertion process is very similar to finding variant entries in the tree. When looking for variant nodes, indicator variables are not bound. However, since the substitution to be inserted must be normalised, the test for variant nodes succeeds in matching two identical indicator variables. A heuristic is used for descending into the tree. It has to handle three different situations: First of all, the heuristic has to select a variant subnode of the current node for descending if such a variant exists. Second, the heuristic selects

a non-variant subnode, which yields a non-empty MSCG, if a variant could not be found. Third, if neither a variant nor a subnode that yields a non-empty common generalisation could be found, the heuristic has to select the empty tree for insertion. In this case the insertion function creates a new leaf node.

Example 7. Insertion of the substitution $\{x \rightarrow f(c, g(d))\}$ to tree A in figure 2 employs the above-mentioned heuristic. As the substitution to be inserted actually is a variant of the root of the tree, the heuristic has to select the subnode of the tree where the insertion process is to be continued. Assume that the node marked with $\{x_1 \rightarrow *1, x_2 \rightarrow g(b)\}$ is selected. Because this node is not suited for the insertion of the new substitution a new intermediate node containing the generalization of this selected node and the new substitution has to be created. The resulting tree B in figure 3 contains a new leaf node and a new inner node marked with $\{x_2 \rightarrow g(x_3)\}$. This substitution is the most specific common generalisation of the substitutions $\{x_1 \rightarrow *1, x_2 \rightarrow g(b)\}$ and $\{x_1 \rightarrow c, x_2 \rightarrow g(d)\}$.

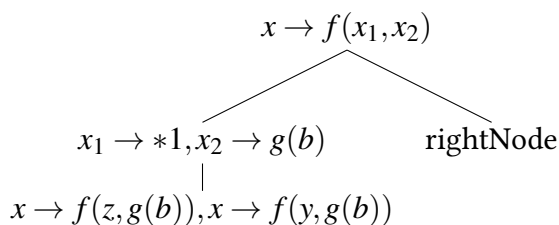


Figure 2: Tree A

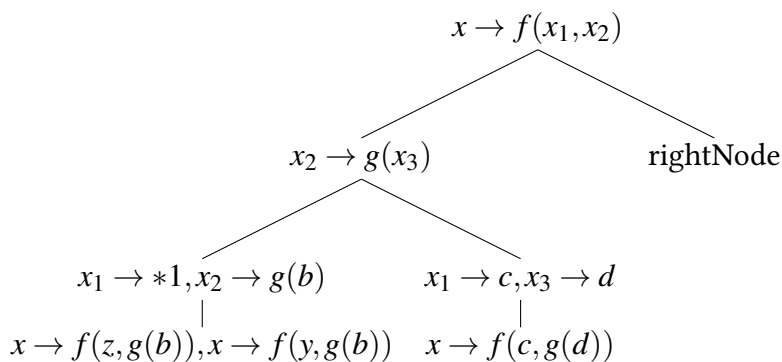


Figure 3: Tree B

4 Instance tries

This chapter presents a new term indexing data structure, called the instance tries (Prokosch & Bry 2020). Unlike other term indexing data structures, instance tries do not rely on heuristics but do consider the partial order of terms. As this term indexing data structure is completely new, it had not yet been considered in experiments.

Goal of this bachelors' thesis is to implement instance tries in Rust, conduct various experiments on real-life datasets and show that they outperform existing term indexing data structures.

A key characteristic of instance tries is that they rely on a matching-unification algorithm. In the following sections, we will explain the term "unification", as well as the unification problem and its connection to instance tries.

Joerg Siekmann in Unification Theory (1989):

"Most knowledge based systems in artificial intelligence (AI), with a commitment to a-symbolic representation, support one basic operation: **matching of descriptions**. This operation, called unification in work on deduction, is the **addition-and-multiplication** of AI-systems and is consequently often supported by special purpose hardware or by a fast instruction set on most AI-machines."

4.1 Unification algorithms

Unification is one of the key procedures in first-order theorem provers. Most first-order theorem provers use a unification algorithm from Robinson (Hoder & Voronkov 2009). Robinson published several unification algorithms, the widely known one requires exponential space, while others require quadratic space. Although the complexity of the well-known one is exponential in the worst case, the algorithm is easy to implement and examples on which it may show exponential behaviour are believed to be atypical. More sophisticated algorithms, such as the (Martelli & Montanari 1982) or the (Paterson & Wegman 1978) algorithm, offer polynomial or even linear space complexity but are harder to implement (Hoder & Voronkov 2009). Very little is known about the practical performance of unification algorithms in theorem provers: Previous case studies have been conducted on small numbers of artificially chosen problems and have compared term-to-term unification while the best theorem provers perform

set-of-terms-to-term unification using term indexing, see [section 2](#).

4.1.1 Unifier

A substitution, as mentioned in chapter 3, is an assignment of terms to variables. In essence, the unification problem in first-order logic can be expressed as follows: Given two expressions containing some variables, find, if it exists, a substitution which makes the two expressions equal. The resulting substitution is called an unifier.

Two expressions e_1 and e_2 are **unifiable** iff

$$\exists \lambda : e_1 \lambda = e_2 \lambda$$

The substitution λ is a **most general unifier** of these two expressions, iff for every unifier σ there exists a substitution τ such that

$$\sigma = \lambda \tau$$

Example 1. Let functions f and g , a and b constants, and x and y variables, and consider the two **first order terms** s and t built from these symbols as follows:

$$s = f(x, b) \quad t = f(a, y)$$

The unification problem is whether or not there exist terms, which can be substituted for the variables x and y in s and t so that the two terms thus obtained are identical. In this example a and b are two such terms and we write

$$\lambda = \{x \rightarrow a, y \rightarrow b\}$$

4.1.2 Instance

In this section, the term **instance** is explained and why it is important for the instance tries. An instance trie is a hierarchical means of viewing instances, which have a logical relationship between them. Moreover, every instance trie is built by inserting queries into an index, such that the query is an instance of an expression, in the index.

Example 2. Let x, y variables and a, b constants. The following expressions should be added in this order to the at first empty index:

$$f(x,y), f(x,b), f(a,y), f(a,b).$$

In order to determine where every expression will be inserted, we need to keep in mind that every child node should be an instance of its parent node. For example having inserted the first three expressions to the index we get the tree presented in figure 4. We now want to insert the last expression to the index. As $f(a,b)$ is an instance of node $f(x,b)$, the expression will be placed under it and we will get the tree pictured in figure 5.

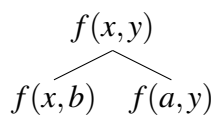


Figure 4: Subtree

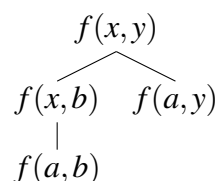


Figure 5: Complete tree

4.1.3 Matching and instantiation

Let λ be a substitution. Often we are looking for **one-sided unification (matching)**:
An expression e_1 is an instance of expression e_2 iff

$$\exists \lambda : e_1 = e_2 \lambda$$

Matching (i. e. One-sided equation solving), as described above, is similar to **instantiation** but not identical. Instantiation does not solve an equation but generates a new expression from an expression, a substitution, and substitution application. Matching is more complicated than substitution application.

4.2 Instance tries and their implementation

In this section, we will define the instance tries and discuss their implementation, as presented in this bachelors' thesis. Key principles of Instances Tries are, first, that they are rooted trees with a variable branching factor. Furthermore, nodes in instance tries hold a logical relationship between them and each node n is labeled with an expression, $expr(n)$. For the mentioned logical relationship between two nodes m and m' of an Instance Trie holds the following:

- If m' is a child of m , then $\text{expr}(m')$ is a strict instance of $\text{expr}(m)$.
- If m is the parent of m' , then $\text{expr}(m)$ is strictly more general than $\text{expr}(m')$.
- If m, n are siblings and m' is an instance of both m, n then m' is a child of the first node. This requires an ordering on expressions in the instance trie.
- Siblings are pairwise incomparable with respect to instantiation and generalisation. **Sibling order** \leq_E makes instance tries unique.

Example 3. We illustrate instance trie indexing with an example set consisting of the following indexing expressions:

$$f(x,y,z), f(x,b,z), f(a,y,z), f(a,c,z), f(a,b,z), f(b,b,z), f(a,y,c).$$

The following tree was produced by inserting each of these expressions into the index.

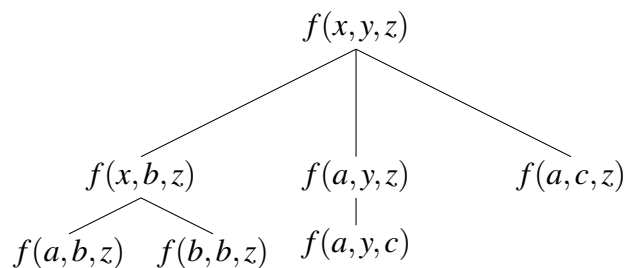


Figure 6: Instance Trie

Let's take the example tree from figure 1 and run the query $f(z,b,z)$.

The results will be the following:

- nodes $f(x,y,z)$ and $f(x,b,z)$ are strictly more general than the query
- nodes $f(a,b,z)$, $f(b,b,z)$ and $f(a,y,z)$ are only unifiable with the query
- nodes $f(a,y,c)$ and $f(a,c,z)$ are not unifiable with the query

Apart from the three orthogonal primitives mentioned above in the example there are 2 more, namely **strict instance** and **variant**. Let's consider two examples for the above-mentioned definitions. Let u,v and x,y variables and a,b constants:

1. The expression $f(a,b)$ is a **strict instance** of the expression $f(u,v)$.
2. The expression $f(x,y)$ is a **variant** of the expression $f(u,v)$.

5 Performance and measurements

The main contribution of this thesis is the comparison of the two index data structure implementations mentioned above, substitution trees and instance tries, based on a supply of real-world benchmarks for indexing. But why do we want to compare these two term indexing data structures?

As mentioned in the book of Peter Graf "Term Indexing" (Graf 1996:177), substitution trees not only provide efficient maintenance and retrieval of terms but also of idempotent substitutions. They achieve maximal search speed paired with minimal memory requirements in various experiments and outperform traditional techniques.

Instance tries, as they are new, could not have been considered in the evaluation of Peter Graf. As mentioned and explained in the chapter "Instance Tries" at [section 4](#), nodes in instance tries, hold a logical relationship between them. Goal of this work, is to run some tests on both substitution trees and instance tries, analyse and compare the result and show that instance tries provide better performance than substitution trees.

5.1 COMPIT (COMParing Indexing Techniques for automated deduction)

For benchmarking, COMPIT (COMParing Indexing Techniques for automated deduction) was used. The main characteristics of COMPIT and reasons for choosing it are as follows:

- COMPIT provides twenty-nine benchmark tests, corresponding to real runs of real systems on real problems.
- COMPIT is produced by various theorem provers, which are Vampire, Fiesta and Waldmeister.
- COMPIT consists of real-life traces made for any requirements of different indexing data structures.
- Index updates (deletions and insertions of terms) are interleaved with (in general far more frequent) retrieval operations.

5.2 Test cases

The method used for creating the benchmarks mentioned above for a given prover is to add instructions making the prover write to a log file a trace each time an operation on the index takes place, and then run it on the given problem. For example, each time a term t is inserted (deleted), a trace like $+t$ (resp. $-t$) is written to the file. It also need to be mentioned that for retrieval operations search of generalisations is required. Moreover, it is needed to store the traces along with information about the result of the operation (e.g., success/failure), which allows detection of cases of incorrect behaviour of the indexing methods being tested. Ideally, there should be enough disk space to store all traces (possibly in a compressed form) of the whole run of the prover on the given problem (if the prover terminates on the problem; otherwise it should run at least for enough time to make the benchmark representative for a usual application of the prover).

The following is an extract from a benchmark file generated by Waldmeister from the TPTP problem LCL109-2. Comments have been added by the authors of the COMPIT benchmarking suite.

```

a/2      # each benchmark file starts with the
b/0      # signature symbols with respective arities
c/1
...
?ab0     # query term a(b,x0), "?" signals failure
?b       # query term b
+ab0     # insert term a(b,x0) to the index
...
!ab5     # query term a(b,x5), "!" signals success
...
-accbb   # delete term a(c(c(b)),b) from index

```

Table 1: An example benchmark file from COMPIT

5.3 Implementation

The main part of the evaluation process is to test a given implementation of indexing on such a benchmark file, using the implementation of the COMPIT-parser. This implementation provides operations for querying and updating the indexing data struc-

ture. The parser first reads the traces, storing them in main memory. After that, every line from the benchmark file is read using PEST grammar. The operations (insertion, deletion, search) are parsed into actions and all terms read from the benchmark file are translated into expressions. The parser provides a translation function for creating expressions for operations to be performed using the given implementation of indexing. Finally, the actions accompanied by the translated terms are applied on the indexing data structure. In order to avoid overheads and inexact time measurements due to translations and reading terms from the disk, the evaluation process first reads the whole benchmark file and then time measurement is switched on as the corresponding sequence of operations is being called. After the final tree has been created and the success or failure of a search operation has been verified, time measurement is switched off.

5.4 Runtime measurements

In order to get a concrete analysis of both term indexing data structures, the runtime of every operation (insertions, deletions, successful searches and failed searches) was measured at the main parts of the respective indexing data structures, as well as the number of unifications and generalisations that occurred for each. Plots were also generated showing the whole procedure of running these benchmarks on the substitution trees and instance tries. The figures below represent the final results of running the benchmarks, using both the substitution trees and instance tries as an indexing technique. The analysis of these results and the conclusion of this work can be found in the chapter "Analysis and Conclusion" in [section 7](#).

6 Code and data

A large part of the work behind this thesis involved writing the source code for the various algorithms described. The implementation consists of the delete and join methods, as well as their tests for the substitution-trees. A big part of the implementation was also the COMPIT-parser and of-course the MSCG algorithm both for terms and substitutions and the associated tests. This work was done with the help and guidance from my supervisor Thomas Prokosch. Rather than including the code in this thesis we decided to make it available for download on the Internet. The various parts of the code and their state of development are listed in Gitlab of the PMS unit at

(<https://gitlab.pms.ifi.lmu.de/loglang>).

7 Analysis

In this chapter some of the results from running the benchmarks on the substitution trees and instance tries are presented and a summary of the observations is given. The bar charts represent the average time in μs needed to run every operation both in substitution trees and instance tries. On the y-axis the time per operation is shown and on the x-axis on the left the instance tries and on the right the substitution trees. In this manner, we can compare the performance of the two data structures. The line charts represent the nodes searched per operation in relation to the time in μs need to run all the operations from the benchmark.

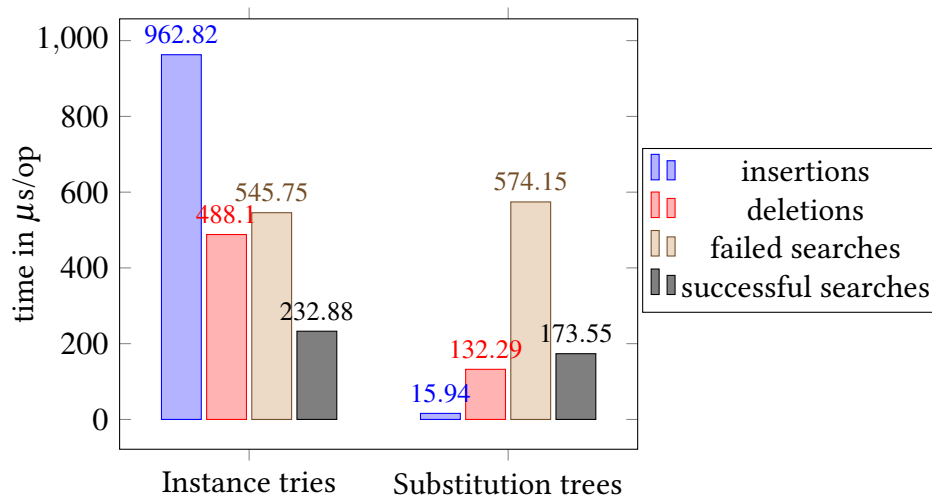
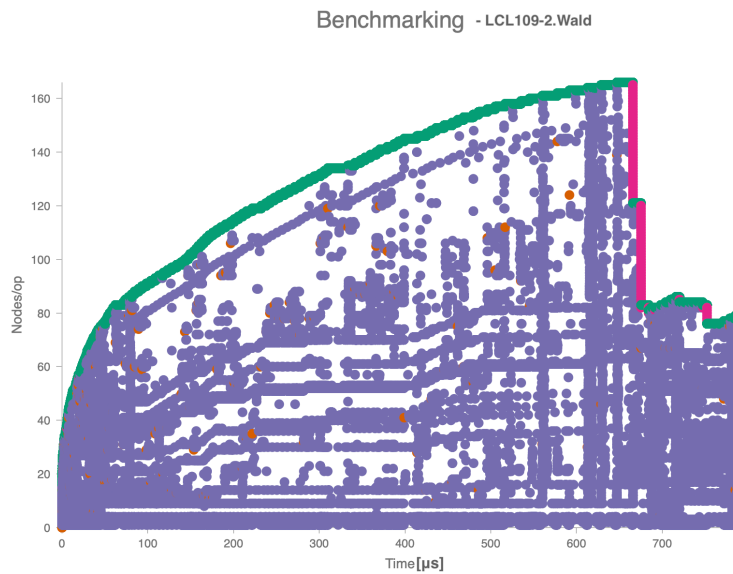


Figure 7: Results - LCL109-2.Wald

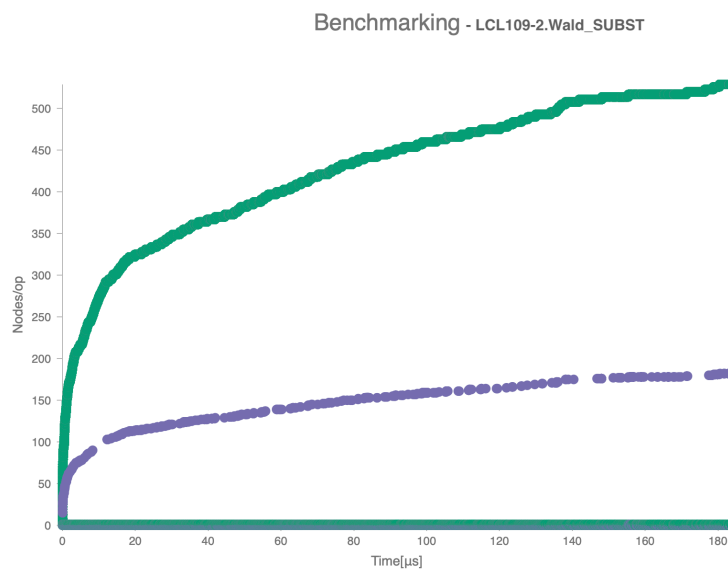
Figure 7 shows the results of running the benchmark LCL109-2.Wald on instance tries and substitution trees. It can be observed that the update operations on average are running slower on the instance tries than on the substitution trees, whereas the retrieval operations have almost the same runtime on both.

Figure 8 shows the plot of running the benchmark LCL109-2.Wald on instance tries up and substitution trees down. The points labeled in green are the failed searches, the purple ones are successful searches, the brown ones are insertions and the pink ones indicate deletions.

Looking at Figure 8 for the instance tries, it can be observed that insertions and suc-



(a) Instance Tries - LCL109-2.Wald



(b) Substitution Trees - LCL109-2.Wald

Figure 8: LCL109-2.Wald - Plots

cessful search operations seem to require a logarithmic time to finish running. Furthermore, if an expression is searched in the tree but not found the majority if not all the nodes of the tree need to be searched. The insertions, on the other hand, don't need to search the whole tree to find the position where the expression has to be inserted.

On the other hand, looking at the plot of the substitution trees, only the failed and successful searches can be observed. This is due to the fact that the update operations don't need to search many nodes in the substitution trees, whereas the retrieval operations reach until 500 nodes. From this figure it can be concluded that the retrieval operations in substitution trees require a logarithmic time.

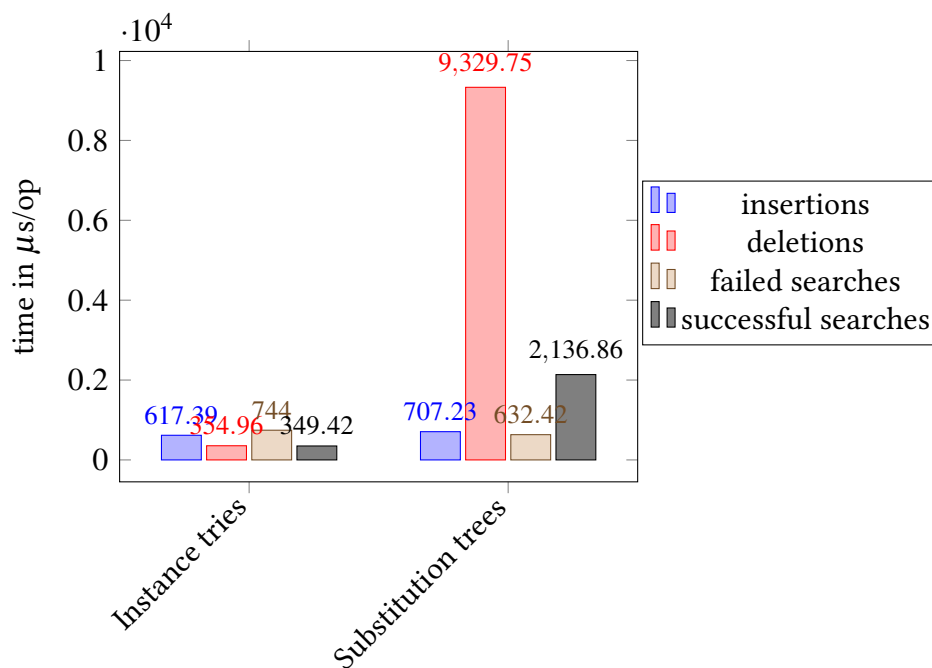


Figure 9: Results - GRP024-5.Wald

Figure 9 presents the results of running the benchmark GRP024-5.Wald on instance tries and substitution trees. It can be observed that the insertions and failed searches have a very similar runtime on the instance tries than on the substitution trees, whereas the successful searches and especially the deletions run much slower on substitution trees. This indicates that further work needs to be done to be able to identify the exception in the deleting operations.

Figure 10 shows the plot of running the benchmark GRP024-5.Wald on instance tries up and substitution trees down. The points labeled in green are the failed searches, the purple ones are successful searches, the brown ones are insertions and the pink ones indicate deletions. A similar pattern can be again observed as in the last example.

Figure 11 was produced by running the benchmark LAT009-1.Wald on instance tries and substitution trees. We can draw the following conclusions: First, the insertions need a bit more time on instance tries than on substitution trees, whereas

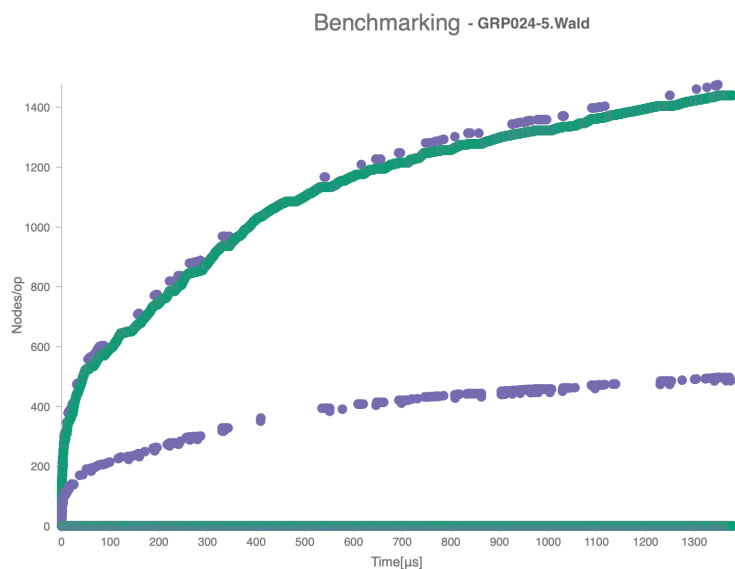
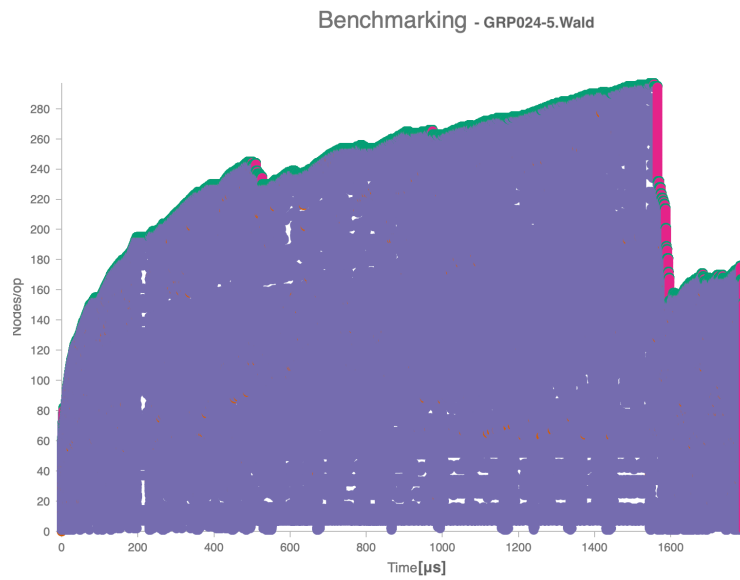


Figure 10: GRP024-5.Wald - Plots

the deletions run slower on substitution trees than on instance tries. The numbers are very close. The update operations run significantly slower on instance tries. It is obvious that the performance of instance tries depends greatly on the unification algorithm used.

Figure 12 presents the plot of running the benchmark LAT009-1.Wald on instance

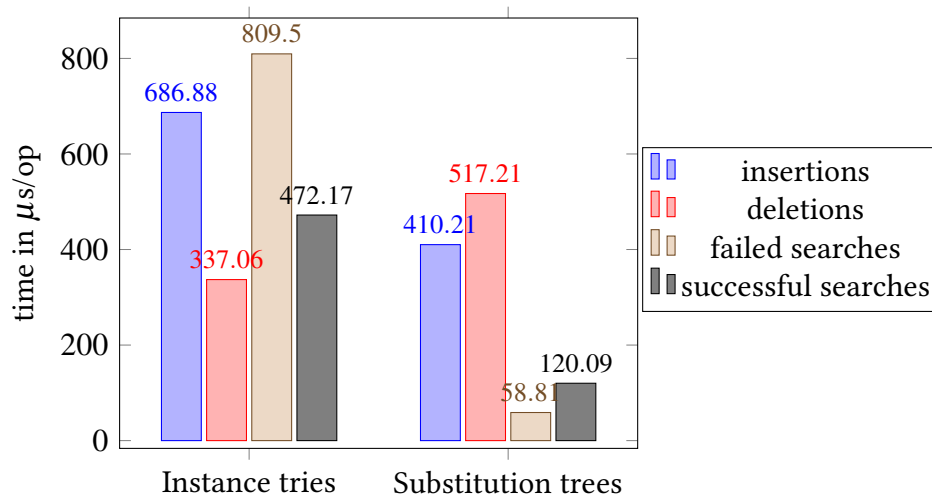
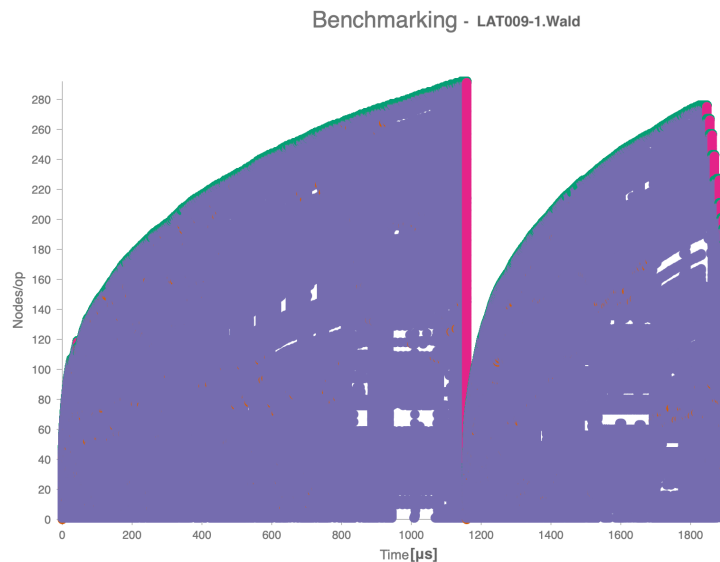


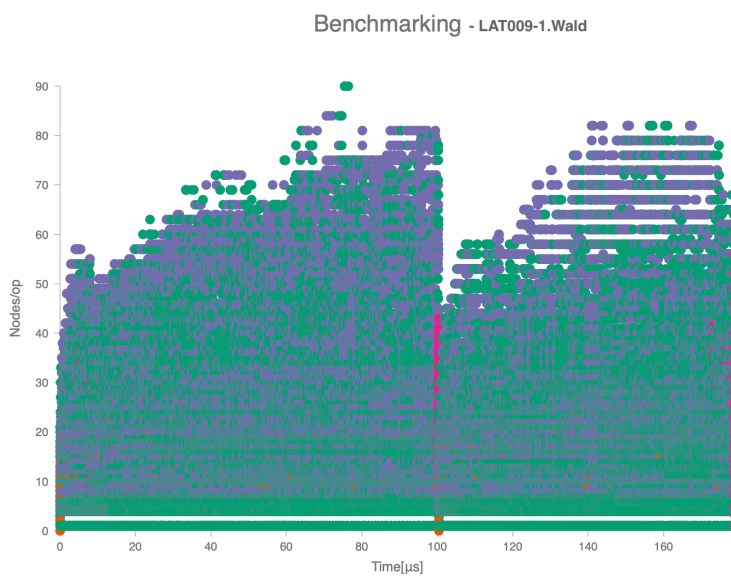
Figure 11: Results - LAT009-1.Wald



(a) Instance Tries - LAT009-1.Wald

tries up and substitution trees down. The points labeled in green are the failed searches, the purple ones are successful searches, the brown ones are insertions and the pink ones indicate deletions. It can be observed that the points presenting update operations are positioned almost at the same place at both plots. Retrieval operations have again a logarithmic growth on instance tries but a different pattern on substitution trees.

The reason why on this chart of the substitution trees we can observe update and retrieval operations and in the other ones not, is because in this example the retrieval



(b) Substitution Trees - LAT009-1.Wald

Figure 12: LAT009-1.Wald - Plots

operations search until 90 nodes of the tree, just like the update operations.

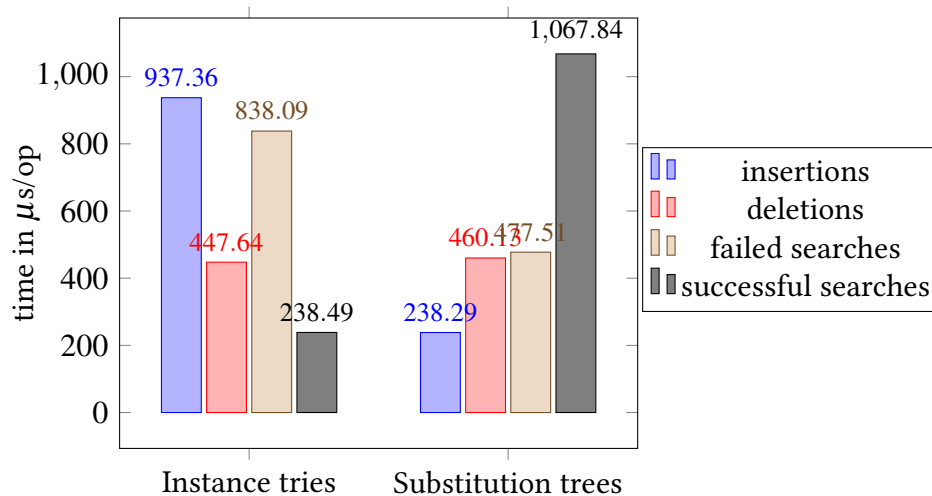


Figure 13: Results - RNG028-5.Wald

The chart shown in Figure 13 was produced by running the benchmark RNG028-5.Wald on instance tries and substitution trees. The interesting observation is that the successful searches in this example need far less time on instance tries than on substitution trees. The deletions need approximately the same time and insertions and failed searches run slower on instance tries. This is also a very positive result for the instance tries as with a bit higher average time per inserting operation, instance tries provide much faster search results. Furthermore, by using instance tries the results are more predictable, while the trees are also smaller, with fewer nodes.

Figure 14 presents the plot of running the benchmark RNG028-5.Wald on instance tries up and substitution trees down. The points labeled in green are the failed searches, the purple ones are successful searches, the brown ones insertions and the pink ones indicate deletions. It can be observed that on the upper plot of the instance tries the points presenting the successful searches are placed mainly at the bottom part of the chart. This is because as mentioned before, in this example the successful searches on instance tries need significantly less time and search fewer nodes than on substitution trees. On the substitution trees, on the other hand, we can observe the same growth as in the first example.

8 Conclusion

From the examples mentioned before and in general after running several benchmarks on both instance tries and substitution trees following conclusions can be drawn:

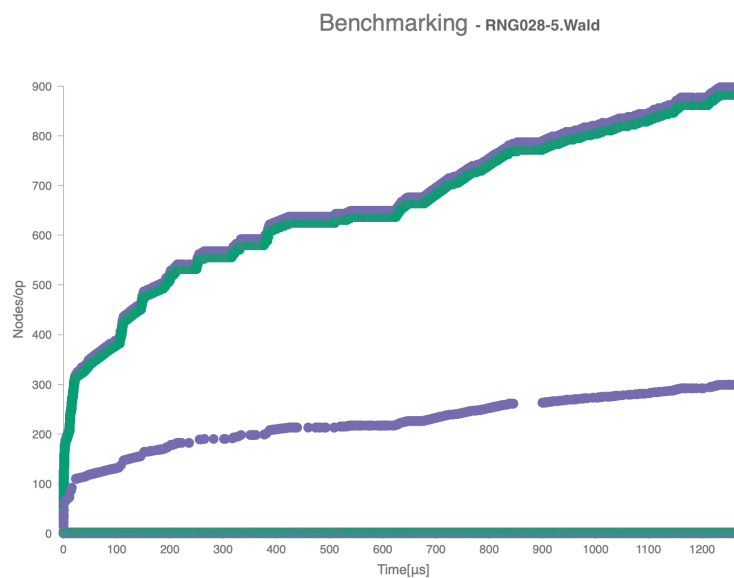
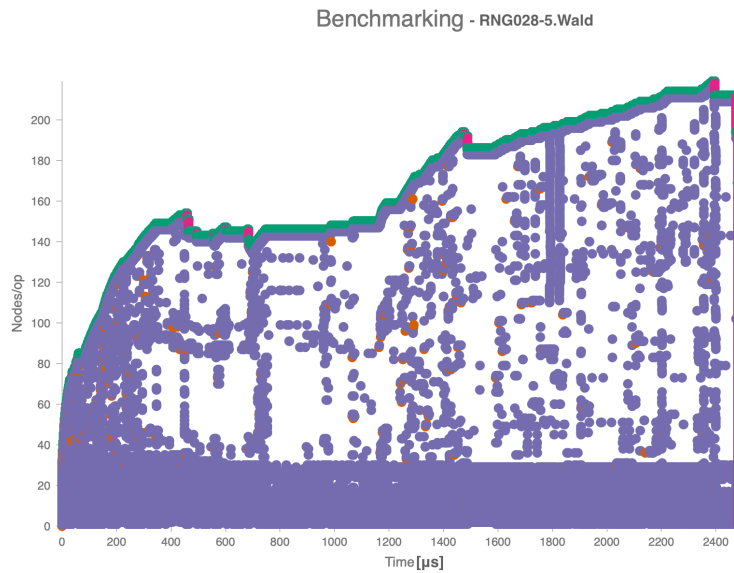


Figure 14: LAT009-1.Wald - Plots

Insertions and successful search operations seem to require logarithmic time to run on instance tries. The same can be observed for the substitution trees but for the retrieval operations, not insertions.

Furthermore, inserting operations on instance tries are usually slower than inserting operations on substitution trees. This was expected as instance tries rely upon unifi-

cation; work to speed up the unification algorithm used is ongoing.

Moreover, a very important feature of instance tries is that every node is inserted at a certain position in the index. Every child node is an instance of its parent node and is placed in the trie based on the variables and constants that the expression in the node contains. This makes the inserting operations slower, as the exact position for every expression should be located in the trie. On the other hand due to this ordering, instance tries provide faster retrieval of expressions. Described in more detail, the term indexing data structures were developed to provide easier and faster retrieval of expressions. At this point, there is no need to search a whole list for example to retrieve an expression. Instance tries were built to enhance even more the performance of retrieval operations, but require a slightly higher insertion time. In particular, they offer predictability and therefore a "sacrifice" needs to be made in average update operation time. This is a positive characteristic of instance tries as there is major need for faster retrievals in term indexing data structures than for faster update operations. Retrieval operations on instance tries are competitive with the ones on substitution trees despite using an unoptimised unification algorithm (Robinson 1976) and relying on expressions, not substitutions, as the core data structure. Instance tries with this unification algorithm can compete substitution trees, which in various experiments present low average retrieval time. This outcome is positive for the new term indexing data structure as the improvement of the unification algorithm used is expected to lead in enhancement of performance and even better results.

According to the conducted results, substitution trees sometimes outperform instance tries. However, as mentioned already, instance tries still rely on expressions and not substitutions as their core data structure and use an exponential unification algorithm. Runtime differs greatly between various theorem provers: Waldmeister runs fast for both instance tries and substitution trees, Fiesta runs fast on instance tries and slowly on substitution trees and Vampire runs slowly on both. For this project examples were derived from Waldmeister theorem prover. The benchmarks consisted of smaller terms than the ones in Fiesta theorem prover. The two term indexing data structures have been tested with benchmarks from the theorem provers Fiesta and Waldmeister.

9 Future work and suggestions

This thesis promoted the overall work until a certain point leading the way for further investigation. In this section a few suggestions for future work will be presented.

Firstly, more tests need to be run on both instance tries and substitution trees to get more results in order to draw reliable conclusions. Furthermore, instance tries need to be implemented not only using expressions but also substitutions as their core data structure. Moreover, apart from the performance, the memory requirements need to be measured as well, with running benchmarks on both instance tries and substitution trees. Lastly, although the benchmarks from COMPIT provide reliable results, it would be advisable to run different benchmarks, where update operations search not only for generalisations, but for variants, instances or terms that unify with the given term.

References

- BENTLEY, JAY. 1995. Introduction. *Term indexing*, 1–16. Springer Berlin Heidelberg. URL https://doi.org/10.1007/3-540-61040-5_10.
- DE CHAMPEAUX, DENNIS. 1986. About the Paterson-Wegman linear unification algorithm. *Journal of Computer and System Sciences* 32.79 – 90. URL <http://www.sciencedirect.com/science/article/pii/0022000086900036>.
- GRAF, PETER. 1996. *Term indexing*. Springer.
- HODER, KRYSSTOF, and ANDREI VORONKOV. 2009. Comparing unification algorithms in first-order theorem proving. *Mertsching B., Hund M., Aziz Z. (eds) KI 2009: Advances in Artificial Intelligence. KI 2009. Lecture Notes in Computer Science, vol 5803*. URL https://doi.org/10.1007/978-3-642-04617-9_55.
- JUNG, RALF; JACQUES-HENRI JOURDAN; ROBBERT KREBBERS; and DEREK DREYER. 2017. Rustbelt: securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*. URL <https://doi.org/10.1145/3158154>.
- MARTELLI, ALBERTO, and UGO MONTANARI. 1982. An efficient unification algorithm. *Transactions on Programming Languages and Systems*. URL <https://doi.org/10.1145/357162.357169>.
- MOZILLA, FOUNDATION. 2016. Rust language. URL <https://research.mozilla.org/rust/>.
- NIEUWENHUIS, ROBERT; THOMAS HILLENBRAND; ALEXANDRE RIAZANOV; and ANDREI VORONKOV. 2001. On the evaluation of indexing techniques for theorem proving. *Automated reasoning*, 257–271. Springer Berlin Heidelberg. URL https://doi.org/10.1007/3-540-45744-5_19.
- PATERSON, MIKE S., and MARK N. WEGMAN. 1978. Linear unification. *Journal of Computer and System Sciences* 16.158 – 167. URL <http://www.sciencedirect.com/science/article/pii/0022000078900430>.
- PROKOSCH, THOMAS, and FRANÇOIS BRY. 2020. Give reasoning a trie. *Joint proceedings of the 7th workshop on practical aspects of automated reasoning (PAAR) and*

the 5th satisfiability checking and symbolic computation workshop (sc-square) workshop, hrsg. von Pascal Fontaine, Konstantin Korovin, Ilias S. Kotsireas, Philipp Rümmer, and Sophie Tourret, *CEUR Workshop Proceedings*, Aug. 2752, 93–108. Aachen: CEUR-WS.org. urn:nbn:de:0074-2752-0. URL <http://ceur-ws.org/Vol-2752/paper7.pdf>.

ROBINSON, JOHN ALAN. 1976. Fast unification. *Theorem Proving Workshop Oberwolfach*.

SIEKMANN, JÖRG H. 1989. Unification theory. *Journal of Symbolic Computation* 7.207 – 274, Unification: Part 1. URL <http://www.sciencedirect.com/science/article/pii/S0747717189800124>.

List of Figures

1	Substitution Tree	11
2	Tree A	13
3	Tree B	13
4	Subtree	16
5	Complete tree	16
6	Instance Trie	17
7	Results - LCL109-2.Wald	21
8	LCL109-2.Wald - Plots	22
9	Results - GRP024-5.Wald	23
10	GRP024-5.Wald - Plots	24
11	Results - LAT009-1.Wald	25
12	LAT009-1.Wald - Plots	26
13	Results - RNG028-5.Wald	27
14	LAT009-1.Wald - Plots	28