INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

Evaluation of the instantiation degree metric for instance tries

René Thome

Bachelor's thesis

Supervisor	Prof. Dr. François Bry
Mentor	Dipl. Ing. Thomas Prokosch
Date of submission	13.04.2022

Abstract

Instance tries have been proposed as a new means of storing expressions for automated reasoning tasks. Maintaining and querying instance tries makes heavy use of unification. Unification algorithms with low worst-case complexity usually make use of involved data structures which introduce non-negligible overhead for simple unification problems.

One solution to the problem is to improve the unification algorithms themselves. Another solution, investigated by this thesis, proposes changes to instance tries which allow to perform unification conditionally, that is when certain (cheap) checks successfully completed. This thesis describes the necessary changes and evaluates them based on a Rust implementation of instance tries.

Zusammenfassung

Instanzbäume wurden als neue Datenstruktur zur Speicherung von Ausdrücken für Automated Reasoning vorgeschlagen. Für das Aufrechterhalten der Ordnung von Instanzbäumen und deren Abfrage muss stark von Unifikation Gebrauch gemacht werden. Unifikationsalgorithmen mit geringer Laufzeit-Komplexität erfordern für gewöhnlich komplexe Datenstrukturen, die einen nicht vernachlässigbaren Mehraufwand für einfache Unifikationsprobleme verursachen.

Eine Lösung dieses Problems besteht darin die Unifikationsalgorithmen selbst zu verbessern. Eine andere Lösung, welche in dieser Arbeit untersucht wird, schlägt eine Anpassung der Instanzbaum-Datenstruktur vor, welche erlaubt Unifikation nur unter bestimmten Bedingungen durchführt, das heißt wenn bestimmte vorgelagerte (einfache) Tests erfolgreich durchgeführt wurden. Diese Arbeit beschreibt die notwendigen Anpassungen und beurteilt sie auf Basis einer Rust-Implementierung der Instanzbäume.

Acknowledgments

I thank Prof. Dr. François Bry for giving me the opportunity to work on this topic and thereby letting me gain experience well outside the regular curriculum of a common Bachelor's degree in computer science. I am also very grateful to my mentor Thomas Prokosch who over the course of this work offered much guidance and gave me excellent advice on academic practice. Due to the global pandemic all work regarding this thesis needed to be done remotely. Therefore I very much appreciate that all relevant processes were adjusted to this new situation by the Research Unit Programming and Modelling Languages. Finally, I thank my family, my friends and my colleagues for their general support which goes well beyond the course of this thesis.

Contents

CHAPTER 1

Introduction

Automated theorem proving and logic programming are areas of research in the field of automated reasoning, which itself can be placed in the much more general area of logical approaches to artificial intelligence [Gra96, 1,2]. Applications in both automated theorem proving and logic programming make use of term indexes which store logical expressions. Depending on the exact application the requirements for such a term index may vary greatly. Generally, automated theorem proving applications operate on highly dynamic and continuously growing indexes, while term indexes used by logic programming applications are rarely changing in comparison. Several term indexing data structures exist. Instance tries have been proposed only recently and aim to fulfill the requirements of both logic programming and automated theorem proving applications.

This introduction introduces concepts and notation required to understand the aim of this thesis. These concepts include logical expressions, substitutions and how finding substitutions for given expressions is related to determining relations between said expressions. Later in this work a method is provided to improve the process of determining the aforementioned relations.

1.1 Expressions and substitutions

The noun *expression* refers to expressions of first-order logic. These expressions are constructed from a countable set of variable symbols **V** and a finite set of non-variable symbols **F**. Non-variable symbols or expression *constructors* are always associated with an arity; following convention, non-variable symbols with zero arity are called *constants*.
Expressions are defined as follows:

1. A variable symbol is an expression.
2. A non-variable symbol with zero arity is an expression.
3. A non-variable symbol with arity $n > 0$ followed by n well-formed expressions the latter of which are separated by commas and enclosed in parentheses, is an expression.

By convention, x, y, z are variable symbols. The letters a, b, c refer to non-variable symbols with zero arity and the letters f, g, h refer to those non-variable symbols with arity greater zero.

Example. The expression $f(g(x), a, h(y, b))$ consists of a *ternary constructor* symbol f, a *unary constructor* symbol g, a *binary constructor* symbol h, two *variable* symbols x, y and two *constants* a, b.

Substitutions

Let $\mathbf{V}$ be the set of variable symbols and $\mathbf{E}$ a set of expressions. A substitution is a total function that maps variable symbols to expressions. The domain of a substitution is the set of unique variable symbols $\{v_1, v_2, .., v_n\} \subseteq \mathbf{V}$ and the range of a substitution is the set of expressions $\{e_1, e_2, .., e_n\} \subseteq \mathbf{E}$; both with $n \in \mathbb{N}_0$. Such a substitution $\sigma : V \to E$ can be written in set notation as $\sigma = \{v_1 \mapsto e_1, v_2 \mapsto e_2, .., v_n \mapsto e_n\}$. The application of a substitution σ to an expression E yields an expression and is noted as $E\sigma$. A bijective substitution with its range being the same as its domain is a *renaming* substitution.

Example. Let $\sigma = \{x \mapsto a, y \mapsto b\}$. The example expression $E_1 = f(g(x), a, h(y, b))$ contains the variable symbols x and y. The application of σ to E_1 yields the expression $f(g(a), a, h(b, b))$. Let $\rho = \{x \mapsto y, y \mapsto x\}$. The substitution ρ is a renaming substitution and its application to E_1 yields the expression $f(g(y)), a, h(x, b))$.

1.2 Unification and relations between expressions

An essential task in both logic programming and automated theorem proving applications is determining whether or not a certain relation is satisfied between given expressions. This section describes these relations.

E_1 is unifiable with E_2

An expression E_1 is unifiable with another expression E_2 if a substitution σ exists such that $E_1\sigma = E_2\sigma$. As becomes apparent from the equation, this relation is symmetric. The expressions $E_1\sigma$ and $E_2\sigma$ are called the common instance of E_1 and E_2 respectively, sometimes also referred to as their unification [Kni89, 94-95].

Example. The expressions $E_1 = f(g(x), a, h(b, b))$ and $E_2 = f(g(a), a, h(y, b))$ are unifiable with the substitution $\sigma = \{x \mapsto a, y \mapsto b\}$. $E_1\sigma = f(g(a), a, h(b, b)) = E_2\sigma$.

E_1 is more general than E_2; E_2 is an instance of E_1

The expression E_1 is more general than the expression E_2 and E_2 is an instance of E_1 if there exists a substitution σ with $E_1\sigma = E_2$.

Example. Let $E_1 = f(x, y)$ and $E_2 = f(a, b)$ be two expressions. The application of the substitution $\sigma = \{x \mapsto a, y \mapsto b\}$ to E_1 yields E_2, showing that E_1 is more general than E_2 or equivalently that E_2 is an instance of E_1.

E_1 is a variant of E_2

The expression E_1 is a variant of E_2 if there exists a *renaming* substitution ρ with $E_1\rho = E_2$. As renaming substitutions are invertible, the variant relation is symmetric. Intuitively, an expression is a variant of another expression if variable symbols in either of the expressions only need to be renamed to yield the respective other.

Example. Let $E_1 = f(x,y)$ and $E_2 = f(y,x)$ be expressions. The application of the *renaming* substitution $\{x \mapsto y, y \mapsto x\}$ to E_1 yields the expression E_2.

Determining whether or not two expressions are unifiable and finding a unifying substitution is a problem solved by unification algorithms. An operation closely related to unification is *matching*.

E_1 matches E_2

The expression E_1 matches the expression E_2 if there exists a substitution μ such that $E_1\mu = E_2$.

While the relations described above are relevant in the context of logic programming and automated theorem proving applications, in the context of the underlying term indexing data structure relations between expressions are categorized slightly different, using the following four mutually exclusive relations:

E_1 is strictly more general than E_2

The expression E_1 is strictly more general than the expression E_2 if there exists a substitution σ' that is *not* a renaming substitution with $E_1\sigma' = E_2$.

E_1 is a strict instance of E_2

The expression E_1 is a strict instance the expression E_2 if there exists a substitution σ' that is *not* a renaming substitution with $E_1 = E_2\sigma'$. Again, this relation is inverse to the previous relation.

E_1 is a variant of E_2

The definition for the variant relation is the same as described above.

E_1 is only unifiable with E_2

The expressions E_1 is only unifiable with the expression E_2 if a substitution σ' exists that is *not* a renaming substitution with $E_1\sigma' = E_2\sigma'$ and E_1 is neither a strict instance of E_2 nor strictly more general than E_2.

Examples. In this thesis, the relations between expressions are abbreviated as shown in this table:

Relation	Infix notation	Example	
variant	VR	$f(x,y)\,\mathrm{VR}\,f(x,z)$	with substitution $\{y \mapsto z\}$
strictly more general	SG	$f(x,y)\,\mathrm{SG}\,f(a,b)$	with substitution $\{x \mapsto a, y \mapsto b\}$
strict instance	SI	$f(a,b)\,\mathrm{SI}\,f(x,y)$	with substitution $\{x \mapsto a, y \mapsto b\}$
only unifiable	OU	$f(a,y)\,\mathrm{OU}\,f(x,b)$	with substitution $\{x \mapsto a, y \mapsto b\}$
not unifiable	NU	$f(a,y)\,\mathrm{NU}\,f(b,z)$	no substitution exists

1.3 Aim of this thesis

The instance trie data structure described in the next chapter makes heavy use of unification and matching to determine relations between expressions, using a matching-unification algorithm to solve both the matching and unification problem. However, this algorithm is compute-intensive despite its optimizations; a simple check that can rule out certain relations without running the algorithm would hence be desirable. This check, as elaborated in Chapter 4, considers the number of variable symbols in both expressions. To illustrate this optimization, consider that it should be determined whether or not expression E_1 is *more general than* expression E_2. If E_1 does not contain variable symbols while E_2 does, E_1 cannot be more general than E_2. The following example shall make this idea more concrete.

Example Assume a term index which stores the expressions $E_1 = f(x,y)$, $E_2 = f(a,b)$ and $E_3 = f(b,b)$. The term index shall be queried for expressions that are *more general than* the expression $E_q = f(a,z)$. The expression E_q contains one variable symbol z, so the two expressions E_2 and E_3 not containing variable symbols cannot be more general than E_q. Only E_1 with two variable symbols may be more general than E_q. While considering the number of variable symbols in the expressions does give an indication as to whether or not the relation may be satisfied, it is not sufficient to show that E_1 is actually more general than E_q. This still needs to be determined by a matching or unification algorithm.

The aim of this thesis is to first formalize a method of using the number of variable symbols in two given expressions to indicate whether or not these expressions might satisfy a desired relation that would otherwise be determined by means of matching. This method is then applied to instance tries and its performance impact is experimentally evaluated.

CHAPTER 2

Term indexing

As stated at the beginning of the introduction, applications in both logic programming and automated theorem proving make use of term indexes to store logical expressions and the requirements to the term index varies greatly between applications in the two fields. The purpose of a term index is to act as a fast interface to a large set of expressions. Fast in the sense that the performance impact on the application itself is minimized because improvements to insertion or retrieval operations on a term index data structure are tied to performance improvements to the applications employing the index [McC92]. Especially in the case of automated theorem proving applications, rapid growth of the index over the application run time causes increased overhead for insertion or retrieval operations and consequently leads to degradation of overall application performance.

Expressions are retrieved from term indexes by issuing queries to it. These queries are issued with an expression as the query key and a query mode that indicates the relation that should be satisfied between this query key and each one of the query results.

Query modes The four query modes a term index for applications in the fields of logic programming and automated theorem proving should support are *more general than*, *instance of*, *unifiable with* and *variant of* [RSV01, 1863]. These four modes are connected to the mutually exclusive relations described in Section 1.2 as follows. Queries with mode *more general than* shall return indexed expressions that are either *variants of*, or *strictly more general than* the query key. Inversely, queries with mode *instance of* shall return indexed expressions that are either *variants of*, or *strict instances of* the query key. Lastly, queries with mode *unifiable with* shall return indexed expressions that are either *variants of*, *strictly more general than*, *strict instances of*, or *only unifiable with* the query key.

One characteristic of term indexes is whether or not their query results include only those expressions that satisfy the query relation or a superset of those expressions which requires subsequent filtering of included false-positive results. A term index is an imperfect filter if false-positives need to be filtered from its query results. If queries to the term index only return those expressions that satisfy the given query relation the term index is called a *perfect filter* [Gra96, 7-8].

2.1 Early approaches to term indexing

Many different approaches for term index data structures emerged over time. Some of the earliest approaches to term indexing were more necessity-driven [McC92], while younger approaches are the product of research on term indexing itself.

Matching pretest. The matching pretest is a simple check that tests whether or not two expressions could match. In essence, this test makes use of the fact that the application of a substitution to an expression E must yield an instance E' of this expression with at least as many symbols as E (see Section 1.1). This test may be used prior to executing more complex methods when testing whether or not two expressions match. If the number of symbols in a suspected instance E_2 is less than that of another expression E_1, no further checks need to be conducted as E_2 cannot be an instance of E_1. This technique requires counting the number of symbols in all expressions, ideally storing this value as an attribute of each expression [Gra96, 44-45]. The matching pretest does not consider variable symbols in expressions and is thus imperfect.

Superimposed codewords Superimposed codewords is an indexing technique that maps attributes of indexed items to fixed-length bit masks. This technique allows the retrieval of indexed items that shall have a set of desired attributes. Using the bitwise OR operation, the bit masks of all desired attributes combined. The index is then searched using this combined bit mask which is prone to errors and thus does not provide a perfect filter for indexed items [Gra96, 47-50].

Path-Indexing. The Standard Path-Indexing method is a set-based indexing technique. Expressions are stored in so-called *path lists*, which are accessed either via a referencing hash table or trie [Sti89].

Discrimination Trees. Discrimination trees store expressions in their leaves while inner nodes refer to prefixes of these indexed expressions. Basic discrimination trees do not differentiate between different variable symbols and are imperfect filters, requiring the query results to be subsequently checked for false-positive results by means of unification or matching [McC92]. There also exist some variations of the basic discrimination tree data structure.

Abstraction Trees. The Abstraction tree data structure stores expressions as substitutions in a tree structure. Indexed expressions are represented by paths from the root to a leaf and are constructed by consecutive application of all substitutions along the path to the root [Ohl90].

Substitution Trees. Substitution trees combine aspects of both discrimination and abstraction trees. As with abstraction trees, expressions are also represented by substitutions along a path from the root to a leaf.

2.2 Instance tries

Instance tries are ordered and thus a deterministic data structure designed to store logical expressions for applications in the field of automated reasoning [PB20a, 93]. Unification and matching are the main operations for updating and querying instance tries, requiring most of the execution time for both retrievals and insertions. Instance tries are used with a

matching-unification algorithm [PB20b]. The actual number of performed matchings and unifications during operations on an instance trie depends on the particular set of expressions stored in the data structure and the exact operation performed on it.

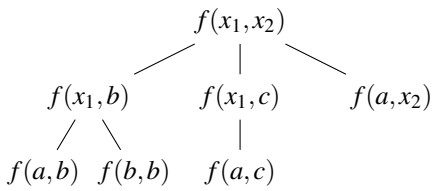
Structure of instance tries

Instance tries are trees which store expressions for term indexing. The structure of instance tries is such that expressions in child nodes must always be strict instances of their parent node. Sibling nodes are ordered which makes instance tries stable in the sense that two instance tries are structurally indistinguishable if they store exactly the same expressions. However, the ordering is not important in this work.

An example shall visualize the concepts of instance tries. Assume that expressions

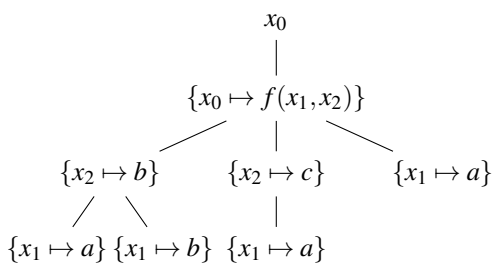
$$f(x_1,x_2),\ f(x_1,b),\ f(x_1,c),\ f(a,x_2),\ f(a,b),\ f(b,b),\ f(a,c)$$

are to be stored in an instance trie. Then, the corresponding instance trie may be visualized (in a simplified manner) as follows:



The previously mentioned relation between parent and child nodes becomes apparent in the above example: All nodes are strict instances of their parent.
For example, the expression $E_2 = f(x_1,b)$ at the first child of the root is a strict instance of the expression $E_1 = f(x_1,x_2)$ at the root. As stated in Section 1.2 this requires a non-renaming substitution σ for which $E_1\sigma = E_2$ holds; i.e. the substitution $\sigma = \{x_2 \mapsto b\}$.

As stated, the visualization above is simplified. Like other trie data structures, also known as prefix trees, expressions stored in an instance trie are not associated with only a single node but rather with a path from the root to a node. The root node of the instance trie data structure carries a variable symbol, all other nodes store substitutions. Expressions stored in an instance trie are retrieved by traversing the path from the root to the desired node, applying the composition of all substitution on this path, to the variable symbol carried by the root [PB20a, 98]. Accordingly, the same instance trie may be visualized more accurately as follows:



Retrieving the expression $f(a,b)$ corresponding to the path to the leftmost leaf from the above tree would correspond to the substitution application:

$$x_0\{x_0 \mapsto f(x_1,x_2)\}\{x_2 \mapsto b\}\{x_1 \mapsto a\}$$

For the rest of this work the simplified representation described above will be used.

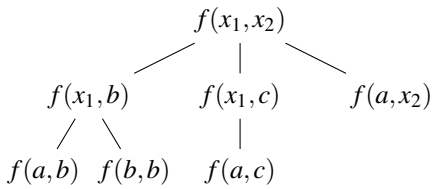
CHAPTER 3

Querying instance tries without optimization

Queries issued to an instances trie consist of an expression as the query key and a query mode as described at the beginning of previous chapter. Indexed expressions satisfying the given query are searched in the index by traversal starting at the root. Traversal continues from parent to child node while multiple sibling nodes are traversed in the order imposed by the instance trie. For each visited node the matching-unification algorithm is used to determine whether the expression at this node satisfies the query. The fact that child nodes are strict instances of their parent node is used during traversal. For example when querying for expressions that are more general than the query key, traversal of child nodes may be skipped if their parent node does not satisfy the query.

Example query

The following instance trie shall be used for the query example below:



Generalizations of $f(z,a)$: Executing a query on the given instance trie for expressions *more general* than the query key $E_q = f(z,a)$ only returns the expression at the root $f(x_1,x_2)$ as it is the only indexed expression that satisfies the query. The matching-unification algorithm mentioned in the first paragraph of this section needs to be executed *four* times. The query is performed in the following steps:

1. The relation between the query key E_q and the expression at the root node $f(x_1,x_2)$ needs to be determined. To determine relation between the expressions, the matching-unification algorithm is used. The result of this procedure is that $f(x_1,x_2)$ is *strictly more general* than E_q (in short $f(x_1,x_2)\,\mathrm{SG}\,E_q$). Thus, the expression $f(x_1,x_2)$ is a solution. Consequently, the child expressions need to be checked as well.

2. Proceeding to the first child of the root, the relation between E_q and $f(x_1,b)$ needs to be determined. Again, this is done using the matching-unification algorithm. The result of this operation is that $f(x_1,b)$ is *not unifiable* with $E_1 = f(z,a)$ (in short $f(x_1,b)\,\mathrm{NU}\,E_q$), meaning $f(x_1,b)$ is not a solution. Due to the required relation between child nodes and their parents, all children of the current node must be strict instances. As this node is not *more general* than the query key, the children can also not satisfy the query relation. Consequently, the traversal of the tree continues at the next child of the root, this node's sibling.

3. Proceeding to the second child of the root, the relation between E_q and $f(x_1,c)$ needs to be determined. As with the previous node, using the matching-unification algorithm yields that E_q is also *not unifiable* with $f(x_1,c)$ (in short $f(x_1,c)\,\mathrm{NU}\,E_q$), and therefore no solution. Thus, the child of this node is disregarded and traversal continues at the root's last child node.

4. Determining the relation between expression $f(a,x_2)$ and E_q using the matching-unification algorithm shows that $f(a,x_2)$ and E_q are *only unifiable* (in short $f(a,x_2)\,\mathrm{OU}\,E_q$), meaning $f(a,x_2)$ is not a solution, either.

As stated in the beginning, the query relation is only satisfied by the expression $f(x_1,x_2)$ which is *strictly more general* than $E_q = f(z,a)$. In all four steps the relation between the query key E_q and the respective indexed expression needs to be determined by means of the matching-unification algorithm. The instance relation between parent nodes and their children allows skipping the child nodes in steps 2 and 3 after determining the relations between the query key and the indexed expressions.

CHAPTER 4

Speeding up instance tries with the instantiation degree

As seen in Chapter 3, querying instance tries involves using the matching-unification algorithm to determine whether indexed expressions satisfy the query mode. For each of the nodes in the instance trie whose expression might satisfy the query, the matching-unification algorithm is called once. This chapter describes an additional check which further restricts the number of nodes whose expressions need to be passed to the matching-unification algorithm by examining the structure of expressions and their relations. It is then explained in detail how this check may be used to skip invocations of the matching-unification algorithm during retrievals from the instance trie data structure.

4.1 Instantiation degree

The instantiation degree is a simple metric proposed by Thomas Prokosch [Pro21]. The instantiation degree of an expression is a single integer value which is determined as follows.

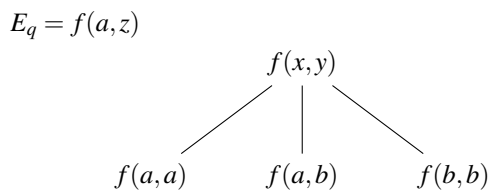
Number of symbols and unique variable symbols in an expression. Let E be an expression. The number of *all* variables and non-variable symbols in E is written as $|E|_S$. The number of *unique* variable symbols is written as $|E|_V$. The following example demonstrates this notation.

Example. In expression $E_1 = f(g(x), a, h(y, b))$ the number of symbols is $|E_1|_S = 7$.
Expression E_1 contains two variable symbols which are x and y, therefore the number of unique variable symbols in E_1 is $|E_1|_V = 2$.
Expression $E_2 = f(g(x), a, h(x, b))$, which is quite similar to expression E_1, also consists of 7 variable and non-variable symbols, i.e., $|E_2|_S = 7$. However, the number of unique variables in E_2 is $|E_2|_V = 1$ because it contains only a single *unique* variable symbol x, occurring twice in the expression.

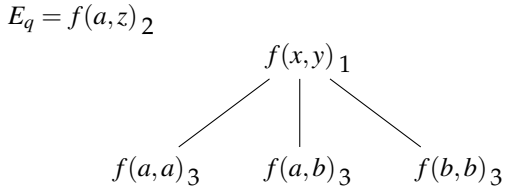
Instantiation degree. Let E be an expression. The instantiation degree for expression E, short $ideg(E)$, is the difference between the number of symbols in E and the number of unique variable symbols in E, i.e. $ideg(E) = |E|_S - |E|_V$.

Example. Expression $E = f(g(x), a, h(y, b))$ contains seven variable and non-variable symbols, i.e. $|E|_S = 7$. E contains two unique variable symbols x and y, i.e. $|E|_V = 2$. Therefore, the instantiation degree of E is 5.

Using the instantiation degree for querying an instance trie. Consider the following instance trie storing the expressions $f(x,y)$, $f(a,a)$, $f(a,b)$, $f(b,b)$. It is queried for expressions that are *more general than* the query key $E_q = f(a,z)$.



Similar to the example of the previous chapter, the query for expressions more general than $E_q = f(a,z)$ only yields the expression $f(x,y)$ which is stored at the root node but still requires four executions of the matching-unification algorithm, one for each node in the instance trie. This means that 3 out of 4 invocations of the matching-unification algorithm yield a negative result. The success rate can be improved by using the instantiation degree. For this, the instantiation degree needs to be calculated once for each node in the tree as well as for the query key. The result of these computations are visualized as subscripts in the following representation of the tree:



It can be observed that the instantiation degree at the root node is less than those at its child nodes which directly follows from the structure of instance tries in which child nodes are instances of their parent nodes and thus contain more non-variable symbols and possibly also more variable symbols than their ancestors. This idea is being formalized in the next section.

To finish this example which aims at retrieving expressions that are more general than E_q, only the expressions of nodes with an instantiation degree of less than or equal to 2 need to be compared with the query key using the matching-unification algorithm. This means that the matching-unification algorithm only needs to be invoked once namely for the expression $f(x,y)$ at the root of the tree, whose instantiation degree is 1.

Hypothesis. The observation above leads to the hypothesis that a comparison of instantiation degrees of two expressions makes it unnecessary to invoke the matching-unification algorithm for certain queries and certain nodes.

4.2 Using the instantiation degree to skip invocations of the matching-unification algorithm

Let E_1 and E_2 be two expressions. The instantiation degree is not sufficient to detect whether E_1 is strictly more general than E_2, a strict instance of E_2, or a variant of E_2. However, it can detect the opposite, i.e. it can detect those cases where E_1 is *not* strictly more general than E_2, *not* a strict instance of E_2, or *not* a variant of E_2. This allows to skip the invocation of the matching-unification algorithm since its negative result can be anticipated by much simpler means: The comparison of two pre-computed integer values expressing the instantiation degree. The following shows for each of these relations in which situations the instantiation degree can be used to make the invocations of the matching-unification algorithm redundant.

E_1 is strictly more general than E_2

E_1 is strictly more general than E_2. Then $ideg(E_1) < ideg(E_2)$.
Proof. E_1 being strictly more general than E_2 requires that a non-renaming substitution σ' must exist for E_1 such that $E_1\sigma' = E_2$. The application of σ' to E_1 cannot, by definition, yield an expression E_2 with less symbols than E_1. Thus, the two cases for the number of symbols that need to be considered are $|E_1|_S = |E_2|_S$ and $|E_1|_S < |E_2|_S$.

a) **Case 1**. In the case of $|E_1|_S$ being equal to $|E_2|_S$ the application of the non-renaming substitution σ' to E_1 results in an expression E_2 with less variable symbols:

$$(E_1 \,\mathrm{SG}\, E_2 \wedge |E_1|_S = |E_2|_S) \Rightarrow |E_1|_V > |E_2|_V \tag{4.1}$$

Consequently, when E_1 is strictly more general than E_2, and both expressions have the same number of symbols, the instantiation degree for E_1 is smaller than the instantiation degree for E_2, which was to show:

$$ideg(E_1) = |E_1|_S - |E_1|_V \overset{4.1}{<} |E_2|_S - |E_2|_V = ideg(E_2) \tag{4.2}$$

Example. The expression $E_1 := f(a, x, g(y))$ is a strict generalization of the expression $E_2 := f(a, a, g(b))$. The required condition $E_1\sigma' = E_2$ holds for the non-renaming substitution $\sigma' = \{x \mapsto a, y \mapsto b\}$. Both expressions E_1 and E_2 have the same total number of symbols $|E_1|_S = |E_2|_S$. The number of unique variable symbols in the first expression $|E_1|_V = 2$ is greater than the number of unique variable symbols in the second expression $|E_2|_V = 0$. From this directly follows the difference in the instantiation degree for the two expressions:

$$ideg(E_1) = |E_1|_S - |E_1|_V = 5 - 2 = 3 < 5 = 5 - 0 = |E_2|_S - |E_2|_V = ideg(E_2)$$

b) **Case 2.** Let E_1, E_2 be expressions. If E_1 is strictly more general than E_2 then there exists a non-renaming substitution σ' such that $E_1\sigma' = E_2$. Further, $|E_1|_S < |E_2|_S = |E_1\sigma|_S$. For the number of unique variables, the following cases emerge:
First, $|E_1|_V = |E_2|_V = |E_1\sigma'|_V$. Then, $ideg(E_1) < ideg(E_2)$ since $|E_1|_S < |E_2|_S$.
Second, $|E_1|_V < |E_1\sigma'|_V$. Then, $ideg(E_1) < ideg(E_2)$ since every unique variable symbol also counts as a symbol, i.e. $|E_1\sigma'|_S \geq |E_1|_S + (|E_1\sigma'|_V - |E_1|_V)$.
Third, $|E_1|_V > |E_1\sigma'|_V$. Then, $ideg(E_1) < ideg(E_2)$ due to the definition of the instantiation degree as the difference between the number of symbols and the number of unique variable symbols in an expression.

Example. The expression $E_1 := f(a,x,g(y))$ is a strict generalization of the expression $E_2 := f(a,h(x,y),g(y))$. The required condition $E_1\sigma' = E_2$ holds for the non-renaming substitution $\sigma' = \{x \mapsto h(x,y)\}$. The number of symbols in the first expression $|E_1|_S = 5$ is less than that in the second expression $|E_2|_S = 6$. For the instantiation degrees follows:

$$ideg(E_1) = |E_1|_S - |E_1|_V = 5 - 2 = 3 < 4 = 7 - 3 = |E_2|_S - |E_2|_V = ideg(E_2)$$

Thus, if an expression E_1 is *strictly more general than* an expression E_2, the instantiation degree of E_1 must be smaller than that of E_2; i.e. $E_1 \,\mathrm{SG}\, E_2 \Rightarrow ideg(E_1) < ideg(E_2)$

E_1 is a strict instance of E_2

The case of an expression E_1 being a strict instance of an expression E_2 is equivalent to E_2 being strictly more general than E_1. Therefore, it follows with the reasoning as above:
$E_1 \,\mathrm{SI}\, E_2 \Rightarrow ideg(E_1) > ideg(E_2)$

E_1 is a variant of E_2

E_1 is a variant of E_2. From this follows that $ideg(E_1) = ideg(E_1)$.
Proof. E_1 being a variant of E_2 can only be satisfied if there exists a *renaming* substitution ρ with $E_1\rho = E_2$. Due to the properties of a renaming substitution ρ, $|E|_S = |E\rho|_S$ and $|E|_V = |E\rho|_V$ must hold for each expression E which means that the number of symbols $|E_1|_S$ must be equal to the number of symbols $|E_2|_S$ and that the number of unique variable symbols $|E_1|_V$ must be the same as the number of unique variable symbols $|E_2|_V$:

$$E_1 \,\mathrm{VR}\, E_2 \quad \Rightarrow \quad (|E_1|_S = |E_2|_S \;\wedge\; |E_1|_V = |E_2|_V) \tag{4.3}$$

As a consequence, the instantiation degree must be equal in both expressions for them to be variants of each other:

$$ideg(E_1) = S_1 - N_1 \overset{4.3}{=} S_2 - N_2 = ideg(E_2) \tag{4.4}$$

Example. The expression $E_1 = f(a,x,g(y))$ is a variant of the expression $E_2 = f(a,y,g(x))$ with $E_1\rho = E_2$ holding for the renaming substitution $\rho = \{x \mapsto y, y \mapsto x\}$. Both expressions have the same total number of symbols and the same unique variable symbols x and y and thus also the same instantiation degree: $ideg(E_1) = 5 - 2 = 3 = ideg(E_2)$

Preventing matching-unification. The observations above show that for three relations *strictly more general than*, *strict instance of*, and *variant of* statements about the instantiation degrees for two expressions can be made. These statements may be used to rule out the existence of relations between two expressions if their respective instantiation degrees are known:

1. If expression E_1 is *strictly more general than* expression E_2 the instantiation degree for E_1 must be smaller than that for E_2. Inversely, if the instantiation degree of E_1 were greater than or equal to that of E_2, the expression E_1 *cannot* be strictly more general than the expression E_2:

$$ideg(E_1) \geq ideg(E_2) \Rightarrow \neg(E_1 \,\mathrm{SG}\, E_2) \tag{4.5}$$

2. On the other hand, if the instantiation degree of E_1 were less than or equal to that of E_2, the expression E_1 *cannot* be a strict instance of the expression E_2:

$$ideg(E_1) \leq ideg(E_2) \Rightarrow \neg(E_1 \,\mathrm{SI}\, E_2) \tag{4.6}$$

3. If an expression E_1 is a *variant of* expression E_2 the instantiation degree for E_1 must be equal to that of E_2. If, however, the instantiation degree of E_1 is different from that of E_2, the *variant of* relation *cannot* be satisfied for the two expressions:

$$ideg(E_1) \neq ideg(E_2) \Rightarrow \neg(E_1 \,\mathrm{VR}\, E_2) \tag{4.7}$$

These observations should make it clear that the instantiation degree may be used to skip invocations of the matching-unification algorithm for pairs of expressions when determining whether or not one expression is *strictly more general than*, *a strict instance of* or *a variant of* another expression.

These mutually exclusive relations are usually not directly used by applications employing a term index. Instead, the query modes *more general than*, *instance of*, and *variant of* are typically used and can be constructed from these mutually exclusive relations as follows:

1. If the instantiation degree of an expression E_1 is greater than that of an expression E_2, using the implications 4.5 and 4.7, the expression E_1 can neither be a variant of, nor strictly more general than the E_2, thus E_1 cannot be more general than E_2.

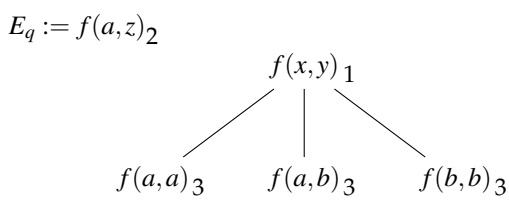
$$ideg(E_1) > ideg(E_2) \Rightarrow E_1 \text{ is not more general than } E_2$$

2. If the instantiation degree of an expression E_1 is less than that of an expression E_2, using the implications 4.6 and 4.7, the expression E_1 can neither be a variant of, nor a strict instance of the E_2, thus E_1 cannot be an instance of E_2.

$$ideg(E_1) < ideg(E_2) \Rightarrow E_1 \text{ is not a strict instance of } E_2$$

By comparing the instantiation degrees of expressions the existence of relations may be ruled out in accordance with the above implications without the need for a matching-unification algorithm. This shall be illustrated by the following example.

Example. Consider again the instance trie from the beginning of this chapter storing the expressions $f(x,y)$, $f(a,a)$, $f(a,b)$, $f(b,b)$. This instance trie is shown in the following figure with the instantiation degree precomputed and annotated as subscripts. Furthermore, the instance trie should be queried for expressions *more general than* the expression $E_q = f(a,z)$ which has an instantiation degree of 2.



Making use of the instantiation degree, the query is processed following these steps:

1. The relation between the query key E_q and the expression at the root node $f(x,y)$ needs to be determined. Comparing the instantiation degree of E_q, which is 2, with the instantiation degree of $f(x,y)$, which is 1, shows that $f(x,y)$ may be more general than E_q because $ideg(f(x,y)) < ideg(E_q)$. To determine the relation between the expressions, the matching-unification algorithm is used. This algorithm gives the result $E_q \,\mathrm{SG}\, f(x,y)$. Thus, the expression $f(x,y)$ is an answer to the query. Consequently, the child expressions need to be checked as well.

2. Proceeding to the first child of the root, the relation between E_q and $f(a,a)$ needs to be determined. Comparing the instantiation degrees of the two expressions shows that $f(a,a)$ cannot be more general than E_q because $ideg(E_q) < ideg(f(a,a))$ meaning $f(a,a)$ is not an answer to the query. Consequently, the traversal of the tree continues at the next child of the root, this node's right sibling.

3. Proceeding to the second child of the root, the relation between E_q and $f(a,b)$ needs to be determined. As with the previous node, comparing the instantiation degrees of the two expressions yields that $f(a,b)$ cannot be more general than E_q, and is therefore no answer to the query. Thus, traversal continues at this node's right sibling, the root's last child node.

4. Proceeding to the last child of the root, the relation between E_q and $f(b,b)$ needs to be determined. As with the previous two nodes, comparing the instantiation degrees of the two expressions yields that $f(b,b)$ cannot be more general than E_q, and is therefore no answer to the query.

While the first step the matching-unification algorithm is used as before to determine the relation between the query key and the expression at the root and the instantiation degree allows to rule out the other three indexed expressions were previously three executions of the matching-unification algorithm were required.

The examples are not representative as they were intentionally kept simple in order to better convey the idea of the instantiation degree. The calculation of the instantiation degrees for each expression and comparing the instantiation degrees prior to conditionally employing the matching-unification imposes additional overhead to the query process. Whether or not this additional overhead introduced by the instantiation degree is compensated by a reduced need for matching-unification is not immediately obvious and needs to be assessed experimentally. This is evaluated in the next chapter.

CHAPTER 5

Evaluation of instance tries with the instantiation degree

An implementation of instance tries in the programming language Rust, as described in [PB20a, 98-102], has been adapted to make use of the instantiation degree. These two versions, the unmodified base version and the enhanced version with the instantiation degree are compared against each other using benchmarks as described in the following section. The degree of instantiation was implemented as a single pointer-sized integer value which is calculated once for each expression upon its insertion.

5.1 Benchmarks

The objective of the benchmarks is to obtain empirical data which allows to make quantitative statements about the usefulness of the instantiation degree for real-world applications using the instance trie data structure. To meet this goal, the experiments of the COMPIT benchmark suite [NHRV01a] have been used. These benchmarks try to provide realistic usage scenarios for term indexing data structures in the field of automated theorem proving which are heavily based on the retrieval of generalizations of expressions. The benchmarks are generated from a selection of problems from the TPTP library [Sut17] being solved by the three provers *Fiesta*, *Waldmeister* and *Vampire* [NHRV01b].
The benchmarks were executed on an Intel Xeon Silver CPU with 16 physical cores, Linux kernel version 4.15 and the implementation of the instance trie data structure was compiled with Rust version 1.58.1. The benchmarks were carried out single-threaded with CPU pinning.

5.2 Performance of the benchmarks

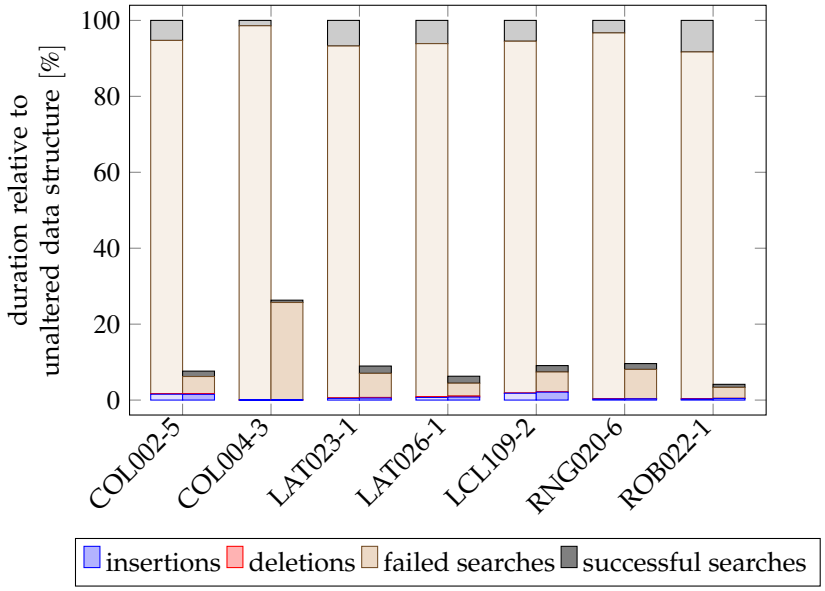
Each of the benchmarks has been executed with and without the instantiation degree and measurements were taken three times to compensate for measuring inaccuracy. The following results are based on the arithmetic mean of the three series of measurements. The raw data of the underlying measurements is listed in the appendix of this work.

Visualization of the results. The goal of quantitatively analysing the impact of the instantiation degree on the instance trie data structure is primarily based on the observation of its impact on the runtime of the benchmarks. To observe the impact of the instantiation degree on the set of benchmarks the following graphs contain a pair of bars per benchmark. The height of each bar refers to the relative duration of the respective benchmark, i.e. relative to the duration of the benchmark on the unaltered data structure.

The colors of the *left* of these bars are *less saturated* and these bars refer to the benchmarks with the *unaltered* data structure and they are therefore always of fixed height (i.e. the relative duration is always 100% of the runtime required with the unaltered data structure). The results of the benchmarks *with* the instantiation degree are depicted in *more saturated* colors on the *right* for each benchmark.

Furthermore, each of the bars is divided into four differently colored segments. The top segment refers to the time required for successful searches, i.e. queries that yield a result. The second segment from the top refers to the time required for failed searches, i.e. queries that yield no result. The third segment refers to the deletion of expressions from the index. Lastly, the bottom segment refers to the insertion of expressions into the index.

Fiesta



Benchmark	Runtime [s]	Insertions [s]	Deletions [s]	Successful searches [s]	Failed searches [s]
COL002-5	12,444.032	191.917	20.629	11,569.877	656.384
COL004-3	1,185.456	0.546	0.015	1,164.048	16.722
LAT023-1	3,720.859	16.502	4.440	3,445.396	250.919
LAT026-1	7,150.140	49.642	14.635	6,644.601	437.399
LCL109-2	2,652.820	47.632	0.786	2,457.692	145.415
RNG020-6	8,966.949	25.738	1.134	8,634.594	295.927
ROB022-1	2,841.971	8.058	0.440	2,594.479	236.071

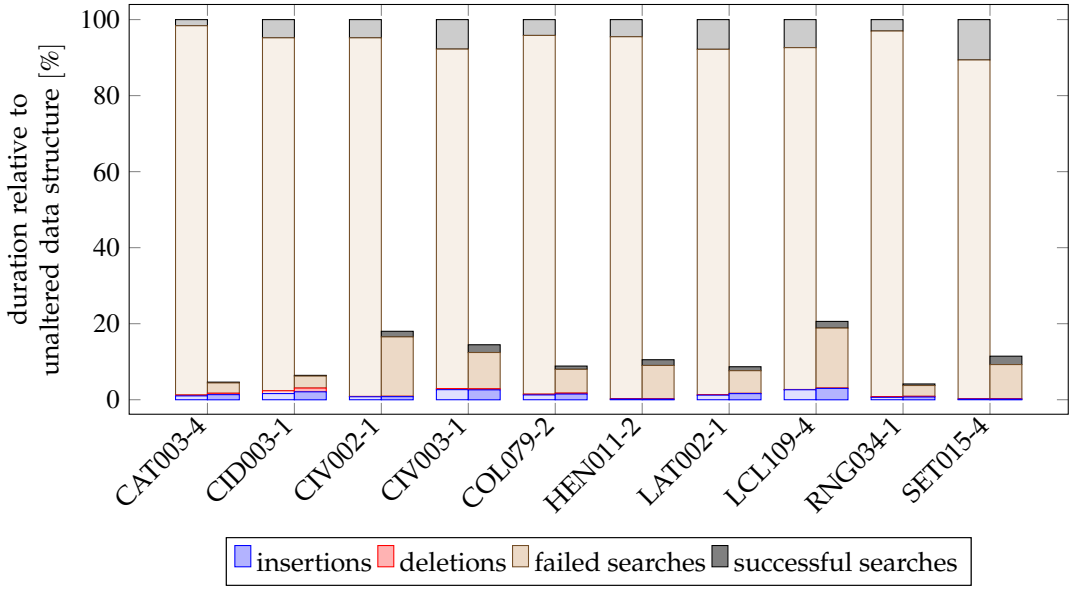
Fiesta measurements without instantiation degree

Benchmark	Runtime [s]	Insertions [s]	Deletions [s]	Successful searches [s]	Failed searches [s]
COL002-5	951.334	188.267	20.048	563.416	176.803
COL004-3	314.495	0.546	0.015	303.044	7.256
LAT023-1	335.581	19.587	5.975	236.096	71.567
LAT026-1	451.068	56.848	18.654	242.267	131.187
LCL109-2	241.891	55.386	0.953	139.838	44.764
RNG020-6	867.716	30.283	1.272	691.777	137.729
ROB022-1	120.416	12.149	0.522	82.297	22.814

Fiesta measurements with instantiation degree

The results for all benchmarks using the *Fiesta* solver show substantial performance improvements. The runtime for all benchmarks has considerably reduced, well below 50% of the runtime that was measured for the unaltered implementation. As expected, the improvements are limited to search operations as the heuristic does not improve the insertion or deletion process. While a certain negative impact on insertions was expected due to the extra effort required to calculate and store the degree of instantiation for each expression, this does not notably show in the over-all benchmark performance.

Vampire



Benchmark	Runtime [s]	Insertions [s]	Deletions [s]	Successful searches [s]	Failed searches [s]
CAT003-4	189,554.977	1,915.564	446.157	184,119.860	3,056.989
CID003-1	51,112.401	829.198	374.924	47,445.786	2,450.878
CIV002-1	95,547.469	748.093	77.594	90,122.280	4,581.384
CIV003-1	127,458.137	3,377.345	353.496	113,822.490	9,886.848
COL079-2	71,644.490	903.276	145.481	67,597.049	2,984.347
HEN011-2	6,729.421	11.797	0.253	6,406.612	303.157
LAT002-1	187,223.703	2,261.872	90.132	170,225.327	14,629.604
LCL109-4	149,078.663	3,908.777	105.386	134,046.923	11,000.888
RNG034-1	78,227.494	498.754	113.261	75,249.820	2,349.657
SET015-4	3,462.511	5.691	0.165	3,081.947	366.838

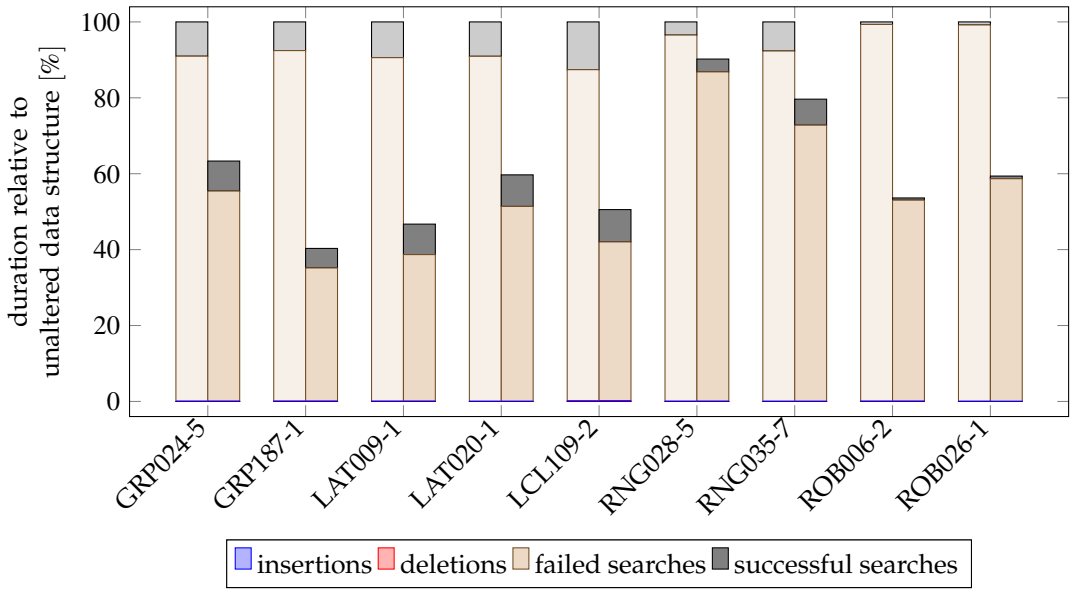
Vampire measurements without instantiation degree

Benchmark	Runtime [s]	Insertions [s]	Deletions [s]	Successful searches [s]	Failed searches [s]
CAT003-4	8,816.730	2,564.817	698.618	5,153.704	390.161
CID003-1	3,294.670	1,072.015	505.067	1,614.816	93.896
CIV002-1	17,226.533	794.560	87.714	14,925.287	1,406.159
CIV003-1	18,464.445	3,331.665	364.989	12,152.035	2,603.414
COL079-2	6,364.711	1,086.882	171.803	4,482.040	615.308
HEN011-2	713.208	12.215	0.247	594.433	101.782
LAT002-1	16,292.127	3,047.948	126.029	11,160.959	1,946.678
LCL109-4	30,751.051	4,463.271	118.791	23,540.935	2,614.035
RNG034-1	3,266.760	576.817	139.455	2,219.917	321.916
SET015-4	401.786	6.453	0.184	313.505	76.232

Vampire measurements with instantiation degree

As with the *Fiesta* solver, there is a substantially positive performance impact of the instantiation degree metric on all *Vampire* benchmarks. While there are also some benchmark related variations, the runtime for all benchmarks was also reduced to well under 50% of the time required with the unaltered implementation.

Waldmeister



Benchmark	Runtime [s]	Insertions [s]	Deletions [s]	Successful searches [s]	Failed searches [s]
GRP024-5	1,040.138	0.168	0.082	938.516	93.145
GRP187-1	5,342.833	1.364	0.300	4,913.643	401.951
LAT009-1	1,192.760	0.191	0.089	1,072.819	111.600
LAT020-1	6,080.174	0.364	0.079	5,494.846	547.072
LCL109-2	110.223	0.064	0.039	95.379	13.751
RNG028-5	1,269.673	0.145	0.060	1,208.215	43.368
RNG035-7	4,732.347	0.274	0.154	4,341.184	360.506
ROB006-2	11,898.472	2.524	0.036	11,757.902	79.390
ROB026-1	7,555.298	0.533	0.002	7,439.903	60.410

Waldmeister measurements without instantiation degree

Benchmark	Runtime [s]	Insertions [s]	Deletions [s]	Successful searches [s]	Failed searches [s]
GRP024-5	663.289	0.212	0.109	571.727	81.554
GRP187-1	2,164.341	1.401	0.326	1,868.232	273.883
LAT009-1	562.711	0.239	0.136	457.991	95.142
LAT020-1	3,651.303	0.450	0.108	3,104.529	502.269
LCL109-2	56.361	0.082	0.056	45.774	9.306
RNG028-5	1,151.701	0.194	0.082	1,086.703	42.330
RNG035-7	3,780.880	0.335	0.195	3,422.532	322.089
ROB006-2	6,410.526	2.966	0.040	6,277.076	68.099
ROB026-1	4,514.220	0.645	0.002	4,401.089	52.421

Waldmeister measurements with instantiation degree

While the results for the *Waldmeister* benchmarks show an improved runtime with the instantiation degree, the positive performance impact is not as substantial as that for the solvers *Fiesta* and *Vampire*. It shows that the additional overhead for insertions introduced by the instantiation degree is negligible even when considering the result with the least improvement.

5.3 Interpretation of the results

The results of the executed benchmarks show a general performance improvement. They show that in all cases the impact of the additional computational resources required to calculate and compare the degrees of instantiation is negligible when compared to all other operations. As stated before, there is no average case for term index usage among all applications in logic programming and automated theorem proving. Therefore an assessment of the impact of this heuristic on the performance of the instance trie data structure may only be achieved by performing benchmarks that simulate such real-world applications. This reflects in the measured results, both in the variation between different problems solved by the same solver and in the different impact on the different solvers. The variation for different problems solved by the same solvers can simply be attributed to different expressions being indexed. The considerable difference between the results for the two solvers Fiesta and Vampire and the results for the Waldmeister solver is certainly tied to how the solver implementations make use of their term index. However, all results indicate that the introduced heuristic does not negatively impact performance and instead mostly leads to a substantial gain in performance.

CHAPTER 6

Conclusion and future work

This work suggests a method for improving the performance of the instance trie term index. To this end, a heuristic, called instantiation degree has been introduced which is solely based on counting the number of variable and non-variable symbols in an expression. The computation of this single integer value can reduce the number of invocations of the matching-unification algorithm which in turn increases the overall performance of instance tries by up to 95.8% for some experiments, and yields gains for all of the experiments that have been conducted. However, given the variety of the underlying problems there is also a variation in the positive performance impact of the introduced heuristic on the instance trie data structure.

Future work on the instantiation degree might be related to conducting further performance tests with granular measurements for a more detailed analysis of the heuristic in the context of instance tries. Likewise, experiments with a focus set on logic programming applications using instance tries may be conceivable. Furthermore, the instantiation degree may also be applicable outside the context of instance tries when matching or matching-unification is used.

Appendix

Raw measurements without instantiation degree

Benchmark	Solver	Run	Runtime [s]	Insertions [s]	Deletions [s]	Failed searches [s]	Succ. searches [s]
CAT003-4	Vampire	1	192,849.540	1,953.113	453.781	187,301.140	3,126.210
CAT003-4	Vampire	2	188,605.360	1,909.252	447.388	183,191.970	3,039.655
CAT003-4	Vampire	3	187,210.030	1,884.326	437.302	181,866.470	3,005.102
CID003-1	Vampire	1	52,103.846	849.405	377.494	48,408.491	2,456.746
CID003-1	Vampire	2	50,483.414	817.251	371.851	46,836.483	2,446.518
CID003-1	Vampire	3	50,749.942	820.939	375.427	47,092.385	2,449.371
CIV002-1	Vampire	1	96,852.011	765.292	78.990	91,333.994	4,656.003
CIV002-1	Vampire	2	95,461.156	754.872	78.905	89,997.285	4,612.371
CIV002-1	Vampire	3	94,329.240	724.114	74.886	89,035.561	4,475.777
CIV003-1	Vampire	1	129,104.690	3,432.011	364.897	115,272.060	10,018.826
CIV003-1	Vampire	2	129,231.500	3,422.691	357.085	115,411.200	10,021.535
CIV003-1	Vampire	3	124,038.220	3,277.333	338.504	110,784.210	9,620.184
COL002-5	Fiesta	1	12,370.638	189.891	20.437	11,504.426	650.688
COL002-5	Fiesta	2	12,341.314	190.088	21.004	11,474.777	650.148
COL002-5	Fiesta	3	12,620.143	195.772	20.446	11,730.427	668.318
COL004-3	Fiesta	1	1,186.105	0.549	0.015	1,164.649	16.723
COL004-3	Fiesta	2	1,181.103	0.544	0.015	1,159.687	16.670
COL004-3	Fiesta	3	1,189.161	0.546	0.015	1,167.807	16.773
COL079-2	Vampire	1	73,146.035	919.789	148.231	69,025.212	3,038.530
COL079-2	Vampire	2	71,293.113	898.839	144.789	67,257.993	2,977.145
COL079-2	Vampire	3	70,494.322	891.200	143.423	66,507.941	2,937.366
GRP024-5	Waldmeister	1	1,041.838	0.168	0.083	940.092	93.291
GRP024-5	Waldmeister	2	1,041.733	0.168	0.082	939.952	93.290
GRP024-5	Waldmeister	3	1,036.843	0.167	0.082	935.503	92.855
GRP187-1	Waldmeister	1	5,305.257	1.362	0.299	4,879.677	398.856
GRP187-1	Waldmeister	2	5,288.072	1.370	0.300	4,863.554	397.306
GRP187-1	Waldmeister	3	5,435.171	1.362	0.302	4,997.697	409.692
HEN011-2	Vampire	1	6,625.111	11.670	0.252	6,306.968	299.107
HEN011-2	Vampire	2	6,862.966	11.946	0.254	6,535.287	308.063
HEN011-2	Vampire	3	6,700.186	11.775	0.252	6,377.581	302.303
LAT002-1	Vampire	1	192,853.820	2,333.143	92.542	175,189.760	15,221.302
LAT002-1	Vampire	2	187,907.690	2,266.985	90.648	170,821.850	14,711.658
LAT002-1	Vampire	3	180,909.600	2,185.487	87.206	164,664.370	13,955.853
LAT009-1	Waldmeister	1	1,192.743	0.190	0.087	1,072.827	111.586
LAT009-1	Waldmeister	2	1,196.292	0.191	0.089	1,075.985	111.949
LAT009-1	Waldmeister	3	1,189.244	0.192	0.090	1,069.646	111.266
LAT020-1	Waldmeister	1	6,046.755	0.362	0.078	5,465.267	544.082
LAT020-1	Waldmeister	2	6,075.674	0.362	0.080	5,491.275	546.518
LAT020-1	Waldmeister	3	6,118.092	0.366	0.079	5,527.998	550.615
LAT023-1	Fiesta	1	3,571.610	16.014	4.399	3,306.466	241.378
LAT023-1	Fiesta	2	3,765.216	16.662	4.446	3,486.697	253.761
LAT023-1	Fiesta	3	3,825.751	16.830	4.476	3,543.024	257.620
LAT026-1	Fiesta	1	7,092.118	49.344	14.946	6,590.765	433.030
LAT026-1	Fiesta	2	7,150.294	49.635	14.423	6,645.553	437.011
LAT026-1	Fiesta	3	7,208.009	49.946	14.535	6,697.486	442.157
LCL109-2	Fiesta	1	2,641.249	47.081	0.776	2,446.447	145.680
LCL109-2	Fiesta	2	2,664.700	47.887	0.791	2,468.965	145.760
LCL109-2	Fiesta	3	2,652.512	47.929	0.790	2,457.666	144.806
LCL109-2	Waldmeister	1	112.323	0.066	0.040	97.182	14.011
LCL109-2	Waldmeister	2	109.113	0.063	0.039	94.418	13.613
LCL109-2	Waldmeister	3	109.234	0.063	0.039	94.537	13.628
LCL109-4	Vampire	1	152,942.170	3,993.911	107.740	137,520.900	11,303.114

Benchmark	Solver	Run	Runtime [s]	Insertions [s]	Deletions [s]	Failed searches [s]	Succ. searches [s]
LCL109-4	Vampire	2	148,042.810	3,890.121	104.802	133,106.680	10,924.457
LCL109-4	Vampire	3	146,251.010	3,842.299	103.614	131,513.190	10,775.092
RNG020-6	Fiesta	1	8,912.315	25.404	1.142	8,582.629	293.793
RNG020-6	Fiesta	2	9,623.073	27.235	1.175	9,270.215	314.648
RNG020-6	Fiesta	3	8,365.460	24.576	1.085	8,050.938	279.340
RNG028-5	Waldmeister	1	1,257.887	0.144	0.060	1,196.941	42.916
RNG028-5	Waldmeister	2	1,282.935	0.145	0.059	1,220.998	43.906
RNG028-5	Waldmeister	3	1,268.197	0.146	0.060	1,206.707	43.282
RNG034-1	Vampire	1	80,798.308	523.057	115.701	77,683.147	2,459.723
RNG034-1	Vampire	2	77,011.631	488.422	112.450	74,095.421	2,299.525
RNG034-1	Vampire	3	76,872.544	484.783	111.632	73,970.892	2,289.722
RNG035-7	Waldmeister	1	4,689.757	0.272	0.155	4,302.212	357.326
RNG035-7	Waldmeister	2	4,716.484	0.273	0.155	4,326.727	359.213
RNG035-7	Waldmeister	3	4,790.801	0.277	0.153	4,394.613	364.977
ROB006-2	Waldmeister	1	12,371.292	2.628	0.036	12,224.409	82.601
ROB006-2	Waldmeister	2	11,719.294	2.486	0.036	11,581.194	78.243
ROB006-2	Waldmeister	3	11,604.831	2.460	0.034	11,468.104	77.325
ROB022-1	Fiesta	1	2,837.674	8.035	0.433	2,590.106	236.198
ROB022-1	Fiesta	2	2,831.161	8.052	0.438	2,584.829	234.939
ROB022-1	Fiesta	3	2,857.079	8.087	0.449	2,608.503	237.077
ROB026-1	Waldmeister	1	7,583.870	0.536	0.002	7,468.457	60.592
ROB026-1	Waldmeister	2	7,620.333	0.536	0.002	7,504.110	60.841
ROB026-1	Waldmeister	3	7,461.691	0.529	0.002	7,347.143	59.797
SET015-4	Vampire	1	3,457.195	5.649	0.164	3,076.675	366.798
SET015-4	Vampire	2	3,452.067	5.675	0.164	3,072.470	365.830
SET015-4	Vampire	3	3,478.272	5.749	0.166	3,096.695	367.885

Raw measurements with instantiation degree

Benchmark	Solver	Run	Runtime [s]	Insertions [s]	Deletions [s]	Failed searches [s]	Succ. searches [s]
CAT003-4	Vampire	1	9,057.266	2,631.359	711.719	5,301.221	403.229
CAT003-4	Vampire	2	8,772.713	2,546.725	691.764	5,135.690	389.139
CAT003-4	Vampire	3	8,620.209	2,516.367	692.370	5,024.201	378.114
CID003-1	Vampire	1	3,424.944	1,097.553	520.335	1,700.109	97.855
CID003-1	Vampire	2	3,214.476	1,054.804	495.913	1,563.275	91.546
CID003-1	Vampire	3	3,244.590	1,063.688	498.954	1,581.063	92.288
CIV002-1	Vampire	1	17,791.231	823.054	91.232	15,389.466	1,475.254
CIV002-1	Vampire	2	16,960.640	775.361	86.306	14,701.418	1,384.129
CIV002-1	Vampire	3	16,927.728	785.264	85.603	14,684.976	1,359.094
CIV003-1	Vampire	1	19,416.429	3,462.407	384.935	12,800.101	2,756.983
CIV003-1	Vampire	2	18,338.709	3,322.506	362.018	12,054.272	2,586.934
CIV003-1	Vampire	3	17,638.197	3,210.083	348.015	11,601.731	2,466.324
COL002-5	Fiesta	1	921.977	181.289	19.768	547.715	170.451
COL002-5	Fiesta	2	953.837	189.362	20.012	564.107	177.554
COL002-5	Fiesta	3	978.187	194.150	20.364	578.426	182.403
COL004-3	Fiesta	1	313.913	0.559	0.015	302.461	7.257
COL004-3	Fiesta	2	314.953	0.538	0.015	303.512	7.256
COL004-3	Fiesta	3	314.619	0.542	0.015	303.160	7.256
COL079-2	Vampire	1	6,660.954	1,126.011	178.881	4,699.722	647.126
COL079-2	Vampire	2	6,226.106	1,067.864	168.598	4,380.612	600.665
COL079-2	Vampire	3	6,207.073	1,066.773	167.931	4,365.786	598.135
GRP024-5	Waldmeister	1	664.103	0.212	0.108	572.528	81.739
GRP024-5	Waldmeister	2	663.113	0.215	0.110	571.560	81.548
GRP024-5	Waldmeister	3	662.651	0.210	0.109	571.094	81.376
GRP187-1	Waldmeister	1	2,143.230	1.392	0.321	1,850.142	271.197
GRP187-1	Waldmeister	2	2,189.751	1.402	0.334	1,890.320	277.152
GRP187-1	Waldmeister	3	2,160.042	1.410	0.323	1,864.233	273.298
HEN011-2	Vampire	1	684.913	11.283	0.228	571.155	97.795
HEN011-2	Vampire	2	720.551	12.525	0.253	600.452	102.839
HEN011-2	Vampire	3	734.162	12.836	0.261	611.691	104.711
LAT002-1	Vampire	1	17,000.035	3,147.120	130.736	11,660.437	2,050.624
LAT002-1	Vampire	2	16,121.093	3,010.848	124.393	11,044.976	1,930.512
LAT002-1	Vampire	3	15,755.253	2,985.877	122.959	10,777.465	1,858.897
LAT009-1	Waldmeister	1	559.468	0.238	0.135	455.485	94.671
LAT009-1	Waldmeister	2	560.808	0.238	0.136	456.491	94.800
LAT009-1	Waldmeister	3	567.856	0.240	0.136	461.996	95.956
LAT020-1	Waldmeister	1	3,658.011	0.454	0.109	3,110.291	502.910
LAT020-1	Waldmeister	2	3,620.117	0.445	0.108	3,078.082	498.248
LAT020-1	Waldmeister	3	3,675.781	0.453	0.108	3,125.214	505.649
LAT023-1	Fiesta	1	328.895	19.076	5.940	231.211	70.300
LAT023-1	Fiesta	2	335.006	19.402	5.999	235.776	71.513

Benchmark	Solver	Run	Runtime [s]	Insertions [s]	Deletions [s]	Failed searches [s]	Succ. searches [s]
LAT023-1	Fiesta	3	342.842	20.282	5.986	241.300	72.886
LAT026-1	Fiesta	1	419.140	52.151	17.162	226.060	121.725
LAT026-1	Fiesta	2	463.122	58.683	19.138	248.462	134.676
LAT026-1	Fiesta	3	470.941	59.710	19.664	252.278	137.161
LCL109-2	Fiesta	1	230.381	52.421	0.924	133.130	42.976
LCL109-2	Fiesta	2	247.815	56.897	0.969	143.298	45.702
LCL109-2	Fiesta	3	247.476	56.841	0.967	143.086	45.614
LCL109-2	Waldmeister	1	58.398	0.085	0.058	47.432	9.635
LCL109-2	Waldmeister	2	55.345	0.081	0.055	44.952	9.135
LCL109-2	Waldmeister	3	55.339	0.081	0.055	44.939	9.148
LCL109-4	Vampire	1	31,555.231	4,552.491	121.065	24,169.738	2,697.243
LCL109-4	Vampire	2	30,701.687	4,448.496	118.630	23,502.537	2,617.439
LCL109-4	Vampire	3	29,996.235	4,388.827	116.679	22,950.529	2,527.424
RNG020-6	Fiesta	1	815.131	27.082	1.179	649.188	131.139
RNG020-6	Fiesta	2	895.826	31.868	1.322	714.601	141.410
RNG020-6	Fiesta	3	892.192	31.899	1.315	711.542	140.638
RNG028-5	Waldmeister	1	1,149.172	0.193	0.083	1,084.546	42.243
RNG028-5	Waldmeister	2	1,154.009	0.194	0.084	1,088.841	42.425
RNG028-5	Waldmeister	3	1,151.923	0.194	0.080	1,086.724	42.323
RNG034-1	Vampire	1	3,446.161	606.534	145.276	2,343.429	341.725
RNG034-1	Vampire	2	3,180.625	562.253	136.056	2,161.509	312.322
RNG034-1	Vampire	3	3,173.494	561.665	137.033	2,154.813	311.702
RNG035-7	Waldmeister	1	3,768.641	0.335	0.194	3,412.008	321.250
RNG035-7	Waldmeister	2	3,788.884	0.336	0.195	3,429.566	322.746
RNG035-7	Waldmeister	3	3,785.115	0.334	0.195	3,426.022	322.273
ROB006-2	Waldmeister	1	6,672.076	3.099	0.041	6,533.904	70.686
ROB006-2	Waldmeister	2	6,415.508	2.970	0.040	6,281.817	68.195
ROB006-2	Waldmeister	3	6,143.993	2.828	0.039	6,015.508	65.417
ROB022-1	Fiesta	1	117.003	11.634	0.511	80.168	22.195
ROB022-1	Fiesta	2	118.624	12.081	0.514	81.110	22.360
ROB022-1	Fiesta	3	125.620	12.732	0.542	85.612	23.887
ROB026-1	Waldmeister	1	4,500.363	0.643	0.002	4,388.046	52.292
ROB026-1	Waldmeister	2	4,526.207	0.645	0.002	4,412.150	52.585
ROB026-1	Waldmeister	3	4,516.091	0.647	0.002	4,403.070	52.388
SET015-4	Vampire	1	399.669	6.220	0.180	311.717	76.165
SET015-4	Vampire	2	401.983	6.450	0.185	313.707	76.220
SET015-4	Vampire	3	403.707	6.689	0.186	315.090	76.313

Bibliography

[Gra96] Peter Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Artificial Intelligence*. Springer, Berlin, Germany, 1996.

[Kni89] Kevin Knight. Unification: A multidisciplinary survey. *ACM Comput. Surv.*, 21(1):93–124, March 1989.

[McC92] William McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *J. Autom. Reason.*, 9(2):147–167, oct 1992.

[NHRV01a] Robert Nieuwenhuis, Thomas Hillenbrand, Alexander Riazanov, and Andrei Voronkov. Let's compit! `https://people.mpi-inf.mpg.de/~hillen/compit/`, 2001. Accessed: 2022-02-01.

[NHRV01b] Robert Nieuwenhuis, Thomas Hillenbrand, Alexander Riazanov, and Andrei Voronkov. On the evaluation of indexing techniques for theorem proving. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning*, pages 257–271, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[Ohl90] Hans Jürgen Ohlbach. Abstraction tree indexing for terms. In *Proceedings of the 9th European Conference on Artificial Intelligence*, ECAI'90, page 479–484, USA, 1990. Pitman Publishing, Inc.

[PB20a] Thomas Prokosch and François Bry. Give reasoning a trie. In Pascal Fontaine, Konstantin Korovin, Ilias S. Kotsireas, Philipp Rümmer, and Sophie Tourret, editors, *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*, volume 2752 of *CEUR Workshop Proceedings*, pages 93–108, Aachen, 2020. CEUR-WS.org. urn:nbn:de:0074-2752-0.

[PB20b] Thomas Prokosch and François Bry. Unification on the run. In Temur Kutsia and Andrew M. Marshall, editors, *The 34th International Workshop on Unification (UNIF'20)*, number 20-10 in RISC Report Series, pages 13:1–13:5, Linz, Austria, June 2020. Research Institute for Symbolic Computation, Johannes Kepler University.

[Pro21] Thomas Prokosch. Personal communication, January 2021.

[RSV01] I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. Term indexing. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1853–1964. Elsevier and MIT Press, 2001.

[Sti89] Mark E. Stickel. The path-indexing method for indexing terms. 1989.

[Sut17] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.