

Term-Indexing for the Beagle Theorem Prover

Timothy Clarence Richard Cosgrove

A subthesis submitted in partial fulfillment of the degree of
Bachelor of Science (Honours) at
The Department of Computer Science
Australian National University

October 2013

© Timothy Clarence Richard Cosgrove

Typeset in Palatino by \TeX and $\text{\LaTeX} 2_{\epsilon}$.

Except where otherwise indicated, this thesis is my own original work.

Timothy Clarence Richard Cosgrove
21 October 2013

For Dana.

Acknowledgements

Thanks to my supervisor, Peter Baumgartner, for all his help and support. Also coffee.

Abstract

x

Contents

Acknowledgements	vii
Abstract	ix
1 Introduction	1
1.1 Goals	1
1.2 Motivation	1
1.3 Structure of this Report	1
1.3.1 Background	1
1.3.2 Method	1
1.3.3 Results	1
2 Background	3
2.1 First Order Logic Terms and Notation	3
Logic Expressions	3
Logic Calculus	3
Soundness and Completeness	4
Positions	4
Substitutions	4
2.1.1 The Superposition Calculus	5
2.2 Automated Reasoning and Theorem Proving	5
2.3 Term Indexing	6
Top Symbol Hashing	6
Discrimination Trees	6
Path Indexing	7
2.4 Fingerprint Indexing	7
2.4.1 Term Fingerprints	7
2.4.2 The Fingerprint Index	8
2.4.3 Comparing Fingerprints and Retrieving	9
2.4.4 Position Variants	10
2.4.5 Why Fingerprint Indexing?	10
2.5 The <i>Beagle</i> Theorem Prover	10
2.5.1 Hierarchic Reasoning	11
2.5.2 Weak Abstraction	11
2.5.3 Rule Based Inference System	12
2.5.4 <i>Beagle's</i> Major Shortcomings	13
2.6 Tools Used	14

Scala	14
Eclipse	14
3 Implementing Fingerprint Indexing	15
3.1 Structure of <i>Beagle</i>	15
3.1.1 Syntax and Data Structures	15
3.1.2 Main Inference Procedure	17
3.2 Building the Fingerprint Indexer	18
3.2.1 Objects and Data Types	19
Positions	19
Fingerprint Features	19
Term Fingerprints	19
Fingerprint Index	19
3.2.2 Building Term Fingerprints	20
3.2.3 Adding Terms to the Index	21
3.2.4 Retrieving Compatible terms	23
3.2.5 Matching with Subterms	24
3.2.6 Current Problems and Term Traces	25
Term Alienation	25
Term Tracing	26
3.2.7 Unit Testing with ScalaTest	26
3.3 Adding Indexing to <i>Beagle</i>	26
3.3.1 Attaching a Fingerprint Index	26
3.3.2 Indexing Superposition	27
From Case	28
Into Case	29
3.4 Extending the Fingerprint Index for Simplification	29
3.4.1 <i>Beagle's</i> Simplification Process	29
Negative Unit Simplification	30
Demodulation	30
3.4.2 Generalising our Fingerprint Index	31
3.4.3 Applying new Indices to Simplification	33
3.5 Tailoring to <i>Beagle's</i> Hierarchic Superposition with Weak Abstraction Calculus	34
3.5.1 The Extended Hierarchical Unification Table	34
3.5.2 Other Tailored Optimisations	36
Pure Background Terms	36
Maximal Literals	37
4 Results	39
4.1 Instrumenting Beagle	39
4.2 Problem Selection	40
4.3 Indexing Metrics	41
4.3.1 Total Run Time	41

4.3.2	False Positives	42
4.3.3	Runtime Per Inference	43
4.4	Comparing Versions of <i>Beagle</i>	43
4.4.1	Totalled Performance Against Indexing Metrics	44
	Total Time	44
	False Positives	45
	Time Per Inference	46
4.4.2	Results for Small Problems	47
4.4.3	Results for Large Problems	48
	Most Significant Superposition Results	48
	Most Significant Negative Unit Simplification Results	49
4.5	Comparing Various Fingerprint Sampling Positions	50
5	Conclusion	55
5.1	The Benefits of Term Indexing for the <i>Beagle</i> Theorem Prover	55
5.2	Future Improvements	55
5.2.1	More Thorough Testing	55
5.2.2	Extended Data Structures	55
5.2.3	More Fingerprint Indices	55
5.2.4	Extensions to Fingerprint Indexing	55
	Symbol count / Other Features	55
	Retrieval Caching	55
	Dynamic Fingerprinting	56
5.2.5	Comparing Indexing Techniques	56
5.3	Final Thoughts	56
A	Result Tables for TPTP Selection	57
A.0.1	Metrics	57
A.0.2	Problem list	57
A.1	Unmodified <i>Beagle</i>	58
A.2	<i>Beagle</i> with Superposition and Simplification Indexing	59
A.3	Including Optimisations Tailored to the Hierarchic Superposition with Weak Abstraction Calculus	60
B	Results when Varying Fingerprint Sample Positions	61
B.1	Sample 1: FP3W	62
B.2	Sample 2: FP4M	63
B.3	Sample 3: FP6M	64
B.4	Sample 4: FP7	65
B.5	Sample 5: FP8X2	66
	Bibliography	67

Introduction

1.1 Goals

1.2 Motivation

- Describe beagle
- Advantages of beagle
- drawbacks
- some instrumentation

1.3 Structure of this Report

In this report we will...

1.3.1 Background

1.3.2 Method

1.3.3 Results

Background

2.1 First Order Logic Terms and Notation

This thesis focuses around the extension of *beagle*, a *first order logic* (FOL) theorem prover. Specifically *beagle* operates on first order logic *with equality*, where the only true *predicates* (boolean functions) are $=$ (the equality operator), \vee (logical OR), and \neg (negation). Other predicates (including \wedge , the logical AND operator) can be defined in terms of these three operations. In order to understand *beagle*'s purpose and functions a basic understanding of first order logic with equality is required; but for reference a brief glossary of important terminology used throughout the paper is provided here.

Logic Expressions

- **Variable:** A general symbol which may be replaced by another term.
- **Function Symbol:** Functions with an *arity* describing how many arguments they take. Arity 0 functions are referred to as *constants*.
- **Term:** An expression made up entirely of function applications and variables.
- **Subterm:** A term contained within another term. $s[u]$ denotes that u is a subterm of s .
- **Equation:** an *equality* statement, such as $s = t$ where s and t are terms. Equations can be used to *emulate* other predicates by adding reserved constants for *true* and *false*. For example $isEmpty(s) = true$.
- **Literal:** An equation or a *negated* equation. For example $\neg(s = t)$, written $s \neq t$.
- **Clause:** Throughout this paper we use clause to refer specifically to a *conjunctive normal form* clause, which is a collection of literals joined with \vee .

Logic Calculus

A *calculus* refers to a set of rules which can be used to infer new information from a knowledge base; usually by combining two or more clauses.

Soundness and Completeness

A logical calculus may be *sound* and/or *complete*. A calculus is sound if it cannot be used to derive an expression which is not truly a theorem. A calculus is complete if it can be used to derive all true theorems in any logical system. Incomplete calculi can occasionally be useful in special cases; but unsound calculi are generally useless.

Positions

Many concepts in this paper require the ability to precisely point out a specific sub-term/symbol within a term. Thus we introduce a syntax for term *positions*. This sort of syntax is a standard concept in the field of logic; but here we will be using a slightly extended syntax as we will have need to reference positions which do not exist [Schulz 2012].

A *position* is given as a (possibly empty) list of natural numbers. $t|_p$ refers to the subterm of t at position p . The empty position, $t|_\epsilon$, refers to the outermost function or variable of a term. A position $t|_n$ refers to the n^{th} argument of the outermost function, $t|_{n.m}$ to the m^{th} argument of that n^{th} argument; and so on. If the position does not exist in the term we return Nil. Consider the following example:

$$t = f(a, g(a, x, y), b)$$

$$t|_\epsilon = f \quad t|_1 = a \quad t|_{2.2} = x \quad t|_{2.3.3} = \text{Nil} \quad t|_{3.2} = \text{Nil}$$

Substitutions

A *substitution* is a function which replaces one or more variables in an expression with another term. Substitutions are required to define two important relations between expressions:

- **Unification:** expressions s and t are *unifiable* if there exists a substitution σ such that $s\sigma = t\sigma$. Unifiable terms will have a unique *most general unifier* (mgu).
- **Subsumption:** an expression s *subsumes* t if there exists a substitution σ such that $s\sigma = t$. This can also be stated as t is an *instance* of s .

2.1.1 The Superposition Calculus

Beagle's logical calculus, the Hierarchic Superposition with Weak Abstraction Calculus, is based on an existing popular set of inference rules known as the Superposition Calculus. The inference rules are provided here for reference; without explanation since their hierarchic variants will be explored later in detail (see Sections 2.5.3 and 3.3.2). Refer to [Bachmair and Ganzinger 1994] and [Asperti and Tassi 2010] for readings related to the development of the Superposition Calculus.

$$\text{Positive Superposition} \quad \frac{l \approx r \vee C \quad s[u] \approx t \vee D}{(s[r] \approx t \vee C \vee D)\sigma}$$

Where $\sigma = \text{mgu}(l, u)$ and u is not a variable

$$\text{Negative Superposition} \quad \frac{l \approx r \vee C \quad s[u] \not\approx t \vee D}{(s[r] \not\approx t \vee C \vee D)\sigma}$$

Where $\sigma = \text{mgu}(l, u)$ and u is not a variable

$$\text{Equality Resolution} \quad \frac{s \not\approx t \vee C}{C\sigma}$$

Where $\sigma = \text{mgu}(s, t)$

$$\text{Equality Factoring} \quad \frac{l \approx r \vee s \approx t \vee C}{(l \approx t \vee r \not\approx t \vee C)\sigma}$$

Where $\sigma = \text{mgu}(s, l)$

2.2 Automated Reasoning and Theorem Proving

Automated Reasoning is a rapidly growing field of research where computer programs are used to solve formal logic problems; typically stated in first order logic. The most common variety are *resolution* provers. These take as input a knowledge base of theorems and a *query* logic statement. The query can be proven a theorem by adding its *negation* to the knowledge base and repeatedly attempting inference rules until arriving at a contradiction.

Some existing resolution theorem provers include:

- **SPASS** [Weidenbach et al. 1999]
- **Vampire** [Riazanov and Voronkov 1999]
- **E** [Schulz 2002]

2.3 Term Indexing

Term indexing is a technique used to better locate clauses and terms for which inference rules in a calculus will apply. In particular, these indices are used to find all terms which will or are likely to *unify* with, *subsume* or be subsumed by a query term. These relations are typical conditions of first order logic inference rules, for example all rules for the superposition calculus in Section 2.1.1 require some unifier σ . The process of implementing Term Indexing forms the core of this paper. Our goal is to implement a recent technique known as Fingerprint Indexing (see Section 2.4); but for examples and comparison we present here some other techniques. Refer to [Graf and Fehrer 1998] for more detailed specifications of these techniques along with some results comparing them.

Top Symbol Hashing

Top symbol hashing simply compares the outermost symbol of any two terms and checks that they are compatible. For example, for the term $f(a, x)$ it would simply look at the symbol f and present any other term with the top symbol f (or a variable) as compatible. All terms compatible in this manner could be retrieved instantly with a standard single layer hash map.

Top symbol hashing is provided as a simple example; it is very rudimentary and is certainly not in active use by any popular theorem provers.

Discrimination Trees

Discrimination Trees work by storing terms in a tree at a position determined by the structure of each term. Each branch in the tree has a label which is either a function symbol (e.g. f , g) or $*$, which may represent any variable. To place a term in the tree we traverse the term from left to right and follow or create branches according to which symbol we observe. This process is simple. but retrieving from the index is a complex process requiring a backtracking algorithm.

This technique can be extended by keeping track of variable names; as opposed to replacing them all with a $*$. This extension makes Discrimination Trees a *perfect* indexing technique in that anything returned by the indexer is guaranteed to be unifiable [McCune 1990]. While this extension sounds ideal, in practice it can result in a large index which takes a considerable amount of time to retrieve terms. This performance loss is often enough to offset any gain from implementing perfect indexing.

Discrimination Trees have another distinct disadvantage in that their tree data structure can grow arbitrarily large depending on the size and variety of observed terms.

Path Indexing

Path Indexing is currently in active use in the Vampire theorem prover [Riazanov and Voronkov 1999]. It behaves similarly to Discrimination Trees but rather than mapping out entire terms it only maps a selection of *paths*. A path in a term is a lists of function symbols ending in a variable or constant, generated by following an argument of each function application. For example there are two paths in $f(g(h(a), x))$, $[f, g, h, a]$ and $[f, g, x]$.

Terms are stored in a tree structure according to which paths they possess; and retrieval is performed by finding terms in the tree which have compatible paths. This process is not straightforward and requires a backtracking algorithm with repeated use of the costly set intersection operation.

2.4 Fingerprint Indexing

Fingerprint Indexing is a recent technique developed by Schulz [2012], the creator of the E prover. It works by computing a *fingerprint* for each logic term; which can then be compared for unification or match compatibility. Like most of the example techniques above, Fingerprint Indexing is *non-perfect* in that compatible fingerprints will not always imply that the associated terms successfully unify/match. The technique makes up for this by being adjustable to arbitrary levels of precision; ranging between what essentially amounts to Top-Symbol Hashing (See section 2.3) to comprehensive, but slow to compute, term comparisons.

2.4.1 Term Fingerprints

A term's *fingerprint* is a list of *fingerprint features* which indicate what the term looks like at a given position. The 4 possible fingerprint features are:

- f (any function or constant in the current system): The term has an application of f at the given position.
- **A**: The term has a variable at the position.
- **B**: The position does not exist in the term, but can be created by expanding a variable via substitution.
- **N**: The position does not exist and cannot be created.

Term fingerprints are computed with respect to a list of positions; by computing which feature exists at each position. We will now revisit the example from the explanation of term positions (in Section) to show the computation of a fingerprint. In the example $\{f, g\}$ are functions, $\{a, b\}$ are constants and $\{x, y\}$ are variables.

$$t = f(a, g(a, x, y), b)$$

$$\text{positions} = [\epsilon, 1, 2.2, 2.3.3, 3.2]$$

$$t|_{\epsilon} = f \quad t|_1 = a \quad t|_{2,2} = x \quad t|_{2,3,3} = \text{Nil} \quad t|_{3,2} = \text{Nil}$$

$$\text{fp}(t, \text{positions}) = [f, a, \mathbf{A}, \mathbf{B}, \mathbf{N}]$$

2.4.2 The Fingerprint Index

Once a term's fingerprint has been generated we use it to store the term in a tree-like data structure known as the *Fingerprint Index*. This data structure is very reminiscent of a Discrimination Tree (See section 2.3) and works similarly. To add a term to the index we (starting from the root node) follow/create branches labelled with the fingerprint features of the term's fingerprint. Once the fingerprint has run out we store the term in a leaf set. Figure 2.1 presents an example Fingerprint Index; built by sampling positions ϵ , 1 and 2 for a variety of terms.

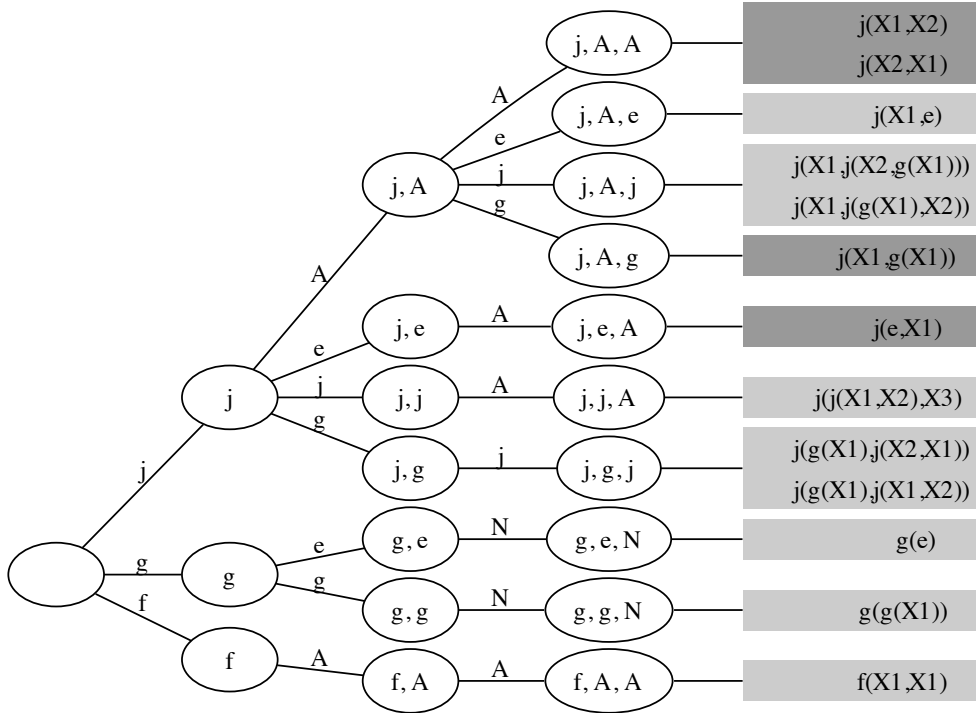


Figure 2.1: Example structure of a Fingerprint Index. Taken unmodified from [Schulz 2012, p7].

The Fingerprint Index has two key advantages which set it apart from the tree data structures used by Discrimination Trees and Path Indexing. First of all, since all term fingerprints are the same length the index has a fixed depth which does not grow overtime. The index's branching factor can increase but this does not add as much complexity as the significant growth of a Discrimination Tree. The second (and most important) advantage is that retrieval is comparatively simple since it only requires the cheap set *union* operation.

2.4.3 Comparing Fingerprints and Retrieving

In order to retrieve terms from a Fingerprint Index we must find all the leaf sets which are *compatible* with the fingerprint of a query term. For two fingerprints to be compatible they need not necessarily be equal; but each of their fingerprint features compared pairwise must be marked with a **Y** in the relevant comparison table. Which table is relevant depends on the substitution operation we are indexing for; Table 2.1 for unification and Table 2.2 for subsumption.

Table 2.1: Fingerprint Feature compare table for Unification (symmetrical). f_1 and f_2 represent arbitrary distinct function symbols. [Schulz 2012, p6]

	f_1	f_2	A	B	N
f_1	Y	N	Y	Y	N
f_2	N	Y	Y	Y	N
A	Y	Y	Y	Y	N
B	Y	Y	Y	Y	Y
N	N	N	N	Y	Y

Table 2.2: Fingerprint Feature compare table for Subsumption (down subsumes across). f_1 and f_2 represent arbitrary distinct function symbols. [Schulz 2012, p6]

	f_1	f_2	A	B	N
f_1	Y	N	N	N	N
f_2	N	Y	N	N	N
A	Y	Y	Y	N	N
B	Y	Y	Y	Y	Y
N	N	N	N	N	Y

As an example consider the terms $j(e, g(x))$ and $j(x, y)$ being compared for unification when sampling positions ϵ , 1 and 2. The fingerprints are $[j, e, g]$ and $[j, \mathbf{A}, \mathbf{A}]$. j is compatible with itself, so the first comparison is marked **Y**. **A** is compatible with any function symbol, so the second and third comparisons are also marked **Y**; and we have that the two terms are potentially compatible for unification.

To retrieve a full set of compatible terms from the Fingerprint Index we generate the query term's fingerprint and proceed to traverse the index tree one fingerprint feature at a time, going down any branch which is compatible with the current fingerprint feature. We proceed down the tree until we reach the leaf sets; at which point we collect them all together with the *union* operation. Note that this is far cheaper than computing set intersections which is required for Path Indexing.

The example Fingerprint Index above (Figure 2.1) actually contains an example of term retrieval; the darker grey leaf sets are all the sets compatible with the query term $j(e, g(x))$.

2.4.4 Position Variants

The performance of Fingerprint Indexing depends greatly on the set of positions sampled to create term fingerprints. Sampling an enormous number of positions will result in almost every term retrieved being truly unifiable; but the Fingerprint Index will be so large and unwieldy that overall performance will steeply decrease.

The key to achieving great performance with Fingerprint Indexing is to balance fingerprint length with retrieval accuracy. Schulz [2012] performed some experiments comparing a variety of position sampling sets. The best performing set sampled 6 positions:

$$\epsilon, 1, 2, 3, 1.1, 1.2$$

As part of the Fingerprint Indexing implementation for *beagle* we will also test a variety of sampling sets; hopefully cross-validating the results from Schulz [2012].

2.4.5 Why Fingerprint Indexing?

In Section 2.3 we described several other techniques for indexing terms, and many more than these few exist. Considering this, it becomes a natural question to ask: why has Fingerprint Indexing been chosen as the focus of this paper?

One major reason is for using Fingerprint Indexing is that it is *new*. Many of the indexing techniques summarised in [Graf and Fehrer 1998] have been around since the 1960s, and all of them have been implemented and thoroughly tested multiple times. Fingerprint Indexing however has (at the time of writing) only ever been implemented by Schulz [2012] as part of his testing. Hopefully by providing a second (from scratch and entirely independent) implementation of Fingerprint Indexing we can confirm the results by Schulz and solidify Fingerprint Indexing as a viable competitor in the field.

Another reason for using this technique is that it is very configurable (in terms of sampling positions); and lends itself to custom optimisations and extensions. For example it is possible to customise the set of fingerprint features by implementing a new comparison table; which could allow more in-depth comparisons in special cases.

2.5 The *Beagle* Theorem Prover

Beagle was developed by Peter Baumgartner et al. of NICTA as a proof of concept, with the intention of demonstrating the capabilities of the *Hierarchic Superposition with Weak Abstraction Calculus* [Baumgartner and Waldmann 2013]. This calculus allows the incorporation of prior knowledge via *background reasoning* modules. These modules act as a sort of ‘black box’ which we can use to rapidly prove facts about background terms; without the need to convert them to an equivalent first order logic formulation. As an example we may think of the background reasoning module as implementing integer arithmetic, in which case any arithmetic theorem like $5 + 5 =$

10 can be proven instantly. In practice these modules can be used to incorporate essentially any collection of facts; but in order to be worthwhile the collection should be extremely large or infinite.

Beagle itself is a resolution theorem prover like the examples above (Section 2.2) which implements this background reasoning calculus. It is intended as a ‘proof of concept’ to demonstrate the capabilities of the Hierarchic Superposition with Weak Abstraction Calculus. This section provides a brief summary of the calculus and its implementation. For a detailed specification of the precise advantages of the calculus refer to [Baumgartner and Waldmann 2013].

2.5.1 Hierarchic Reasoning

Understanding the inference rules of the Hierarchic Superposition with Weak Abstraction Calculus will require some hierarchic reasoning terminology and background knowledge.

Hierarchic reasoning involves two provers, the *foreground* prover and the *background* prover. The foreground prover behaves like a traditional first order logic prover (like the examples in Section 2.2) and uses the calculus inference rules to produce new facts. The background prover exists entirely to implement the background module described above. Each of the provers has its own set of function symbols and variables; and background constants are referred to as *domain elements* (integers in the case of integer arithmetic). Background variables are marked as either *abstraction* or *ordinary*, and background expressions are called *pure* if they do not contain any ordinary background variables.

Hierarchic reasoning system also requires an ordering \succ on terms and literals. This ordering must fulfil a range number of conditions such as well-foundedness (there exists a smallest element), foreground terms are larger than background terms and background terms are larger than domain elements. \succ must also be *total on ground terms*, meaning that any terms s and t which do not contain any variables will have either $s \succ t$ or $t \succ s$.

Note that the logical calculus behind *beagle* is not the first occurrence of using a hierarchy for logical reasoning. A calculus was developed by Bachmair, Ganzinger, and Waldmann [1994] to take advantage of this technique, and Waldmann continued on to co-write the paper outlining Hierarchic Superposition with Weak Abstraction [Baumgartner and Waldmann 2013].

2.5.2 Weak Abstraction

In order to keep the foreground and background reasoning systems segregated it is necessary to clearly split a clause into its foreground and background parts. This is where the process of *weak abstraction* comes in.

In logics with equality there is a general process known as *abstraction*, where a

subterm within a clause may be replaced by a fresh variable.

$$\textbf{Abstraction} \quad \frac{C[t]}{t \not\approx X \vee C[X]}$$

In a hierarchical calculus abstraction can be used to introduce new abstraction variables to take the place of any background subterms. Bachmair, Ganzinger, and Waldmann [1994] extended this form of derivation to what they called *full abstraction*, where abstraction is performed exhaustively until no literal contains both foreground and background operators.

In their recent paper however, Baumgartner and Waldmann [2013] discovered that the process of full abstraction can destroy completeness under particular interpretations. They then go on to propose a new variety of abstraction which they refer to as *weak abstraction*. In weak abstraction only *maximal background subterms which are neither domain elements nor variables* are abstracted. A maximal background subterm is a background subterm not contained in any other background subterm. Abstracted terms are replaced with abstraction variables in the case of pure background terms, or ordinary variables in the case of impure background terms. See the paper itself for weak abstraction examples and details of how this process affects completeness.

2.5.3 Rule Based Inference System

The base inference rules for the Hierarchic Superposition with Weak Abstraction Calculus are essentially identical to the standard superposition calculus; except for the fact that they come with many additional conditions to accommodate background reasoning. These conditions include respecting term/literal orderings and disallowing the use of pure background terms.

The results of any inferences must also have weak abstraction performed on them. This ensures that we only ever have weakly abstracted terms in our knowledge base. The base inference rules follow, taken directly from Section 6 of the Hierarchic Superposition with Weak Abstraction paper [Baumgartner and Waldmann 2013]. See Section 2.1.1 to compare these rules to the original superposition calculus.

$$\textbf{Positive Superposition} \quad \frac{l \approx r \vee C \quad s[u] \approx t \vee D}{\text{abstr}((s[r] \approx t \vee C \vee D)\sigma)}$$

Where (i) $\sigma = \text{simple mgu}(l, u)$, (ii) u is not a variable, (iii) $r\sigma \not\approx l\sigma$, (iv) $t\sigma \not\approx s\sigma$, (v) l and u are not pure background terms, (vi) $(l \approx r)\sigma$ is strictly maximal in $(l \approx r \vee C)\sigma$, and (vii) $(s \approx t)\sigma$ is strictly maximal in $(s \approx t \vee D)\sigma$

$$\textbf{Negative Superposition} \quad \frac{l \approx r \vee C \quad s[u] \not\approx t \vee D}{\text{abstr}((s[r] \not\approx t \vee C \vee D)\sigma)}$$

Where (i) $\sigma = \text{simple mgu}(l, u)$, (ii) u is not a variable, (iii) $r\sigma \not\approx l\sigma$, (iv) $t\sigma \not\approx s\sigma$, (v) l and u are not pure background terms, (vi) $(l \approx r)\sigma$ is strictly maximal in $(l \approx r \vee C)\sigma$, and (vii) $(s \not\approx t)\sigma$ is strictly maximal in $(s \not\approx t \vee D)\sigma$

Equality Resolution

$$\frac{s \not\approx t \vee C}{\text{abstr}(C\sigma)}$$

Where (i) $\sigma = \text{simple mgu}(s, t)$, (ii) s and t are not pure background terms, and (iii) $(s \not\approx t)\sigma$ is strictly maximal in $(s \not\approx t \vee C)\sigma$

Equality Factoring

$$\frac{l \approx r \vee s \approx t \vee C}{\text{abstr}((l \approx t \vee r \not\approx t \vee C)\sigma)}$$

Where (i) $\sigma = \text{simple mgu}(l, s)$, (ii) $r\sigma \not\approx l\sigma$, (iii) $t\sigma \not\approx s\sigma$, (iv) l and s are not pure background terms, and (v) $(l \approx r)\sigma$ is strictly maximal in $(l \approx r \vee s \approx t \vee C)\sigma$

Note the use of a slightly different unification operator, for *simple* mgus. This operator only produces unifiers where abstraction variables are mapped to pure background terms.

Beyond these standard rules the full Hierarchic Superposition with Weak Abstraction Calculus also includes a special rule known as Define; which is required for the calculus to be sufficiently complete. This inference rule does not require searching for unifiers or matchers and is therefore not relevant to term indexing. Refer to [Baumgartner and Waldmann 2013] for its definition and usage.

2.5.4 Beagle's Major Shortcomings

Notice that in the above rules for superposition there are two clauses used to perform the inference. This means that to use it we must search for two clauses which meet all the required conditions. This is obviously a time consuming process; and in the current implementation of *beagle* this search is performed by comparing each possible pair of clauses. This results in an $O(n^2)$ search, which essentially amounts to the *worst case* performance for these inference rules.

The goal of this project is to implement Fingerprint Indexing for *beagle*. This index could be used to collect terms/clauses which are likely to fulfil the unification condition of superposition (condition (i)); significantly restricting the search space required.

Both Equality Resolution and Equality Factoring also require the existence of a simple unifier σ , so it would seem as though Term Indexing is also applicable to these rules. However, these rules only search for unifiable terms within a single clause; which is too small a search space for indexing to be worthwhile.

2.6 Tools Used

Scala

Beagle is written in *Scala*, the Scalable Language. Scala is a functional language and may be confusing to those who are not familiar with the functional programming paradigm. This thesis will contain occasional snippets of Scala code; but note that any snippets used will be accompanied by an explanation and in general an understanding of Scala/functional programming is not required. [École Polytechnique Fédérale de Lausanne (EPFL) 2013]

Eclipse

Making modifications to *beagle* will prove exceedingly difficult without a fully featured development environment. Code for this project was developed in *Eclipse* with the assistance the the *Scala IDE* extension; which allowed simple navigation and error checking [Eclipse Foundation 2013] [Dragos 2013]. Eclipse was integrated with the *ScalaTest* module which allows the creation of simple unit tests; used to ensure code correctness. [Venners et al. 2013]

Implementing Fingerprint Indexing

The Hierarchic Superposition with Weak Abstraction Calculus is a significant achievement for the field of automated reasoning. The theorem prover *beagle* was designed as a proof of concept; and in its current form it is certainly capable of demonstrating the usefulness of the calculus. However, when compared side-by-side with current state of the art provers like E or Vampire (see Section 2.2) it falls short of the mark.

The purpose of this project is to make *beagle* more viable for real-world problems by implementing recent advancements in automated reasoning; particularly in the field of term indexing (see Section 2.3). This chapter covers in detail how Fingerprint Indexing, a recent technique by Schulz [2012] which we described in Section 2.4, was added to *beagle*. This includes analysing the current structure of the *beagle* codebase to determine where indexing can be applied, building the index itself and finally putting the index to active use in simplifying inference rules. The end of this chapter also details a few original improvements to Fingerprint Indexing; designed to tailor it to the nuances of the Hierarchic Superposition with Weak Abstraction Calculus.

3.1 Structure of *Beagle*

Making any extension to the *beagle* project (or any sizeable project for that matter) will obviously require a solid understanding of the existing codebase. This section provides an overview of any existing Scala classes which are relevant to the implementation of the Fingerprint Index; including their structure and any useful existing functionality.

3.1.1 Syntax and Data Structures

Our term indexer must be able to understand the structure of *beagle*'s internal logical objects. Thus the first aspect of *beagle* we must examine is its existing data structures; in particular how it expresses first order logic terms.

Figure 3.1 shows how first-order logic terms are stored. Terms are contained within an Eqn (for Equation) object, which may be ordered (\rightarrow) or unordered ($=$). Equations are then directly passed to a Literal container which stores whether the Equation is positive or negative (true or false). A list of Literals is maintained for

each Clause which are in turn stored in a ClauseSet. These lists are to be interpreted in *Conjunctive Normal Form* (See Section 2.4.1) and thus their ordering is not relevant; save for retrieving specific Clauses / Literals.

This structure is fairly typical and quite directly translates the definitions from Section 2.4.1. The structure could potentially be shortened by removing the Equation object and having Literals directly contain the left and right Terms; but this would be inconsistent with most first-order logic literature and could cause confusion.

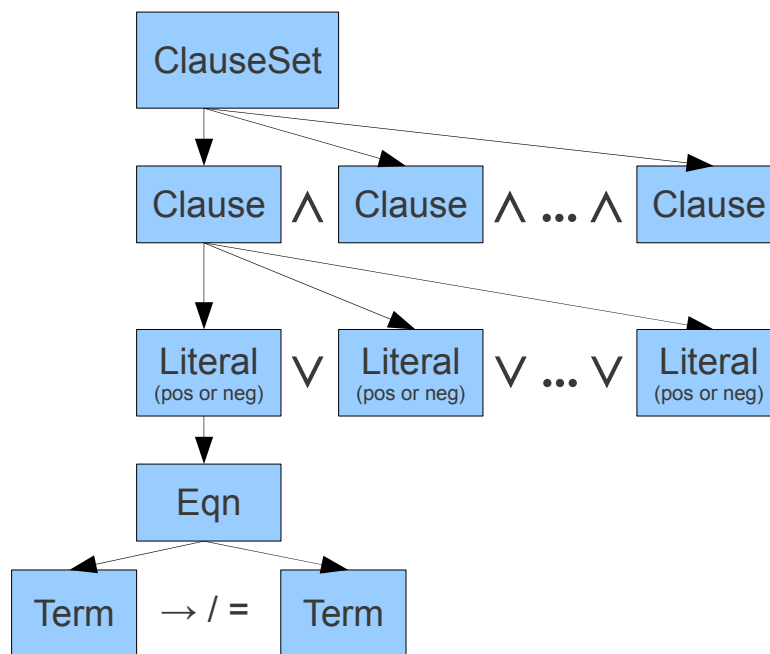


Figure 3.1: Class structure for internal representation of logical formulae.

The object of most concern to our Term Indexer is, naturally, the Term object. We must be able to pull apart Terms in order to sample them at various positions to build indexing fingerprints (see Section 2.4). The Term class itself is actually an *abstract* class with two different primary cases:

FunTerm: Used to express a function application. Consists of a function operator and a (possibly empty) list of arguments. Each argument is another Term object.

Var: Used to express variables. Variables have a *name* and a *sort*, used to identify if the variable is in the foreground or background (i.e. an *abstraction* variable, see Section 2.5.1)

For example, the first-order logic term $f(a, g(x))$ (where x is a foreground variable and other symbols are functions of the appropriate arity) would be expressed as:

$$\text{FunTerm}(f, [\text{FunTerm}(a, []), \text{FunTerm}(g, [\text{Var}(x, FG)])])$$

Knowledge of this structure will be very useful when we move on to constructing Term Fingerprints in Section 3.2.2.

3.1.2 Main Inference Procedure

Now that we have a solid understanding of *beagle*'s relevant data structures we may move on to examining the main loop of the program. This loop repeatedly attempts all the inference rules in the Hierarchic Superposition with Weak Abstraction Calculus (see Section 2.5.3) to generate new information. It also includes some optional rules and optimisations which are not strictly part of the calculus, but drastically increase performance in special cases. This includes:

- **Simplification:** Removes redundant variables and pre-processes some simple clauses. Covered in detail in Section 3.4.
- **The Split rule:** In some cases the Literals of a Clause may be partitioned in two; with each partition consistent with our current knowledge. The Split rule allows us to create a whole new instance of *beagle*'s main loop, to be run in parallel, so that we may consider both options. These 'branches' may be closed if they return unsatisfiable.
- **The Instantiate rule:** Applies to Clauses with background variables in '*finite domains*'. If there are finitely many terms which a variable may represent it is sometimes useful to remove that variable and replace it with one Clause per possible instantiation.

In Listing 1 we present a simplified pseudocode version of the main inference loop. `input ClauseSet` here represents our database of knowledge along with the *negation* of what we are trying to prove. We input the negation since we are ultimately looking for a contradiction (i.e. finding that the set is unsatisfiable); which would then indicate that our original input is always true.

```

new := input ClauseSet
old := empty ClauseSet
While new is not empty
  select := Pop a clause from new
  simpl := Simplify(select, new, old)
  If simpl is a tautology:
    Continue
  If simpl is the empty clause:
    return UNSAT
  If one of the Define, Split or Instantiate rules apply to simpl:
    new := new U ApplicableRule(simpl)
    Continue
  old := old U simpl
  Attempt all inference rules:
    new := new U EqualityResolution(simpl)
    new := new U EqualityFactoring(simpl)
    new := new U Superposition(simpl, old)
end While

```

Listing 1: Pseudocode for *beagle*'s main inference procedure.

Notice the two bolded subroutines in the main procedure. All other routines in the main loop require only the input of *simpl*, but these two also require *old* and/or *new*. This means that their runtime is dependant on the size of the current ClauseSets; and considering that both sets grow overtime these two functions are likely to dominate *beagle*'s runtime.

So *simplification* and *superposition* are the two main areas we should target for improvement with indexing. This is consistent with our earlier analysis of the abstract calculus (see Section 2.5.4) where superposition was identified as the most costly inference rule.

3.2 Building the Fingerprint Indexer

The first step in adding fingerprint indexing to *beagle* is creating the indexer itself; an object which will manage the index and provide functions for adding to it and retrieving from it.

This Section details the creation of the `FingerprintIndex` Scala class; containing all the data types and functions we will need for indexing. This includes building and comparing term fingerprints, addition and retrieval from a complex index structure and any auxiliary functions to assist with these computations.

3.2.1 Objects and Data Types

Here we define in detail any data structures which will be a part of our index. These data structures must be capable of expressing any concepts from the abstract definition of fingerprint indexing, outlined in Section 2.4 and the original paper [Schulz 2012].

Positions

We implement positions in the simple naïve manner, as a list of Integers (where Nil, the empty list, is used to index the top-level term). This directly reflects our position notation given in Section 2.4.1.

Fingerprint Features

Fingerprint features are the four possible symbols we get when sampling a term at an arbitrary position. The meaning of these features is given in Section 2.4.1, so here we provide only the Scala definition. We essentially only require an enumerated type for Fingerprint Features, except for the fact that we must be able to specify a function symbol. Thus we implement this type as four separate case classes implementing an abstract `FPFeature` class; with one of these classes taking a function name as a parameter.

```
1  /** Pseudo enumerated type for fingerprint features */
2  sealed abstract class FPFeature
3  case object FPA extends FPFeature
4  case object FPB extends FPFeature
5  case object FPN extends FPFeature
6  case class FPF(val f : String) extends FPFeature
```

Listing 2: Data type for the 4 Fingerprint Features [Schulz 2012, p5]

Term Fingerprints

With Positions and Fingerprint Features defined it is now very simple to define the Fingerprint for a Term. We take this as simply a list of Fingerprint Features, to be acquired by sampling at various Positions.

Fingerprint Index

The final data structure we require is the actual Index itself, a structure which stores all the indexed terms and their Fingerprints so that they may be later retrieved.

The naïve method for implementing the Index would be to simply use a HashMap from Fingerprints to their corresponding Term. This method would however cause

several problems which would make the correct and efficient retrieval of terms impossible. Fingerprints do not have to be identical in order to match for unification, they will match any Fingerprints which are compatible with respect to some comparison table (see Section 2.4.1). To accommodate this our Index object must store Terms in a way that allows all compatible sets to be collected together. In Listing 3 we present an algebraic data type for an Index, structured as a tree of HashMaps. Each Index is either a collection of Terms (a Leaf) or a mapping from FPFeatures to more Index objects (a Node).

```

1  /** Algebraic Data type for our index. Either we are at a leaf
2      * (set of terms) or must continue traversal via the map. */
3  sealed abstract class Index
4  case class Leaf(set: Set[Term]) extends Index
5  case class Node(map: HashMap[FPFeature, Index]) extends Index

```

Listing 3: Data type for the actual term index. [Schulz 2012, p7]

Note that this Index object does not necessarily take up a significant portion of memory. All Terms are already stored within the ClauseSet object (See Figure 3.1); so the Index itself will generally only add a fairly lightweight structure of pointers.

3.2.2 Building Term Fingerprints

With our required data structures in place we may now begin implementing our Fingerprint Index proper. There are two main components in this implementation: adding to the Index and removing from it. A logical first choice is addition; as we cannot possibly test retrieval until addition is completed. The first step in implementing addition is creating a function to generate Term Fingerprints; by sampling them at a fixed set of positions.

A Term's Fingerprint is a list of Fingerprint Features; which we described in Section 2.4.1 and concretely implemented in Section 3.2.1. Listing 4 provides a function to extract a single Fingerprint Feature from a Term at a given Position; which is the only non-trivial task in Fingerprint generation.

```

1  /** Extract the FPFeature at Position pos of the given Term object. */
2  def extractFeature(term: Term, pos: Position) : FPFeature = pos match {
3    // Reached end of position, check symbol
4    case Nil      => term match {
5      case t:FunTerm => FPF(t.op) // Found function symbol, return it
6      case t:Var     => FPA      // Found variable, return A
7    }
8    case p :: ps => term match {
9      case t:FunTerm => try {extractFeature(t.args(p), ps)}
10     //Non-existent position, return N
11     catch {case e:IndexOutOfBoundsException => FPN}
12     // Found variable BEFORE end of position, return B
13     case t:Var     => FPB
14   }
15 }

```

Listing 4: Scala code to extract fingerprint features for matching.

This code is intended to be a fairly direct implementation of the four fingerprint features described in Section 2.4.1 (and [Schulz 2012]). We simply traverse through the Term until we reach the desired position or we find a variable.

Generating the actual Fingerprint for a Term is now a straightforward process of repeating this function for each desired Position.

3.2.3 Adding Terms to the Index

Now that we can generate Term Fingerprints we must use them to store Term objects at the correct position of our Index data structure (see Section 3.2.1). This is done by following the Index tree mappings for each Fingerprint Feature in the Fingerprint. Traversing the tree is relatively complex; as at each level we may need to create nodes in order to continue traversal. Listing 5 presents code for simultaneously traversing the tree while creating Nodes and Leaves.

```

1  /** Add a Term into the given Index. Traverses Index tree
2   * (adding nodes where needed) and adds Term t to a Leaf set. */
3  private def add (t:Term, fp:Fingerprint, index: Index):Index =
4  (fp, index) match {
5    //Reached a leaf at the end of the Fingerprint. Add to set.
6    case (Nil, Leaf(set)) => Leaf(t::set)
7    //Still traversing tree. Add new Node or Leaf if necessary
8    case (f::fs, Node(map)) => (fs, map.get(f)) match {
9      //Mapping exists. Traverse through it.
10     case (_, Some(index)) => {map += (f -> add(t, fs, index))
11                               Node(map) }
12     //At end of Fingerprint. Create Leaf and add to it
13     case (Nil, None) => {map += (f -> Leaf(List(t)))
14                           Node(map) }
15     //Fingerprint not over. Create Node and continue traversing
16     case (_, None) => {val newIndex:Index = new Node()
17                       map += (f -> newIndex)
18                       add(t, fs, newIndex)
19                       Node(map) }
20   }
21   case (_, Node(_)) => throw new IllegalArgumentException
22     ("Fingerprint is over but we are not at a leaf")
23   case (_, Leaf(_)) => throw new IllegalArgumentException
24     ("Reached a leaf but Fingerprint is not over")
25 }

```

Listing 5: Code to add a Term to the correct Leaf node of the Index data structure defined in Section 3.2.1.

add is a recursive function which moves down the Index one step for each time it is called. Notice that the function takes a Fingerprint as an argument. This argument should initially be the Fingerprint of the input Term (relative to the Index's list of Positions); but with each recursive call of add we strip off one Fingerprint Feature and follow its mapping in the Index. Throughout this process we create new Index Nodes as required; and at the end of the Fingerprint we create (or add to) a Leaf. In accordance with programming best practices the final two cases in the Listing throw meaningful error messages in the case of unexpected input.

We may now index an arbitrary Term object; but it is desirable to be able to index whole Clauses with a single function call. This is done with a straightforward lifting over the expression syntax tree (Figure 3.1) : Clauses index all of their Literals, Literals index their Equation and Equations index each of their two Terms.

3.2.4 Retrieving Compatible terms

Our Index framework is now capable of creating Fingerprints and storing Terms in its pointer structure. The next task in building our index is allowing retrieval of any Terms which are compatible with a query Term; relative to some comparison table.

We will now provide an implementation of the Fingerprint comparison table for unification (Section 2.4.1). To compare two Fingerprints with each other we look at them side-by-side and return *true* or *false* depending on how they match up in the table. The naïve method for performing this check is to simply check for each possible entry in the table manually. However, by examining the table more closely we observe that it can be covered with only four cases:

1. True if the two Features are equal.
2. True if one of the Features is **B**.
3. True if one of the Features is **A**; but the other is not **N**.
4. False otherwise

We may implement these four cases in Scala by using the `match` construct to compare `Set` objects. Using an `unordered Set` object here allows us to take advantage of the unification table being symmetric by checking both directions at once.

```

1  /** Check two Fingerprint features for compatibility based
2     * on the unification table (See page 6 of [Schulz 2012]). */
3  def compareFeaturesForUnification
4     (a:FPFeature, b:FPFeature) : Boolean =
5     (a == b) ||
6     (Set(a,b) match {
7       case x if (x contains FPB) => true
8       case x if (x contains FPA) => !(x contains FPN)
9       case _ => false})

```

Listing 6: Scala implementation of the Fingerprint unification table. [Schulz 2012, p6]

To check whether or not two Fingerprints match is now a simple matter of iterating through the list and checking that each position is a match according to our unification table check. This side-by-side comparison could potentially be improved by employing some variety of hashing function; however this sort of improvement does not apply to our uses for term Fingerprints. Rather than comparing two Fingerprints side-by-side we are only ever interested in retrieving *all* compatible terms from our Fingerprint Index (see Section 2.4.2 and Section 3.2.1).

So, rather than individually comparing the Fingerprints of each indexed Term, we must build a function which traverses the Fingerprint Index tree structure and collects all compatible terms.

```

1  def retrieveCompatible (fp: Fingerprint, index: Index) : TermSet =
2  (fp, index) match {
3  //Collect all compatible (Feature,Index) pairs and continue traversal
4      case (f::fs, Node(map)) =>
5          {for ((k,v) <- map if compare(f, k))
6              yield retrieveCompatible(fs, v)}
7  //Collapse all retrieved sets together with the union operator (:::)
8      .foldLeft (Nil:TermSet) ((a,b) => a ::: b)
9  //Once we reach a Leaf we simply return the compatible set
10     case (Nil, Leaf(set)) => set
11     case (_, Node(_)) => throw new IllegalArgumentException
12         ("Fingerprint is over but we are not at a leaf")
13     case (_, Leaf(_)) => throw new IllegalArgumentException
14         ("Reached a leaf but Fingerprint is not over")
15 }

```

Listing 7: Scala code to collect compatible terms from the index.

We present `retrieveCompatible` (Listing 7); which takes a Term, a Fingerprint and an Index and returns all Terms in the Index which are compatible with respect to the `compare` function. `compare` here is a function implementing a Fingerprint Feature comparison table; such as `compareFeaturesForUnification` from Listing 6. It can be passed to the Fingerprint Index as part of its configuration object (see Section 3.4.2).

`retrieveCompatible` works similarly to `add` from Listing 5; recursively stripping off a Fingerprint Feature and traversing the Index tree. The difference here is that we must ‘branch off’, making a recursive call for each feature which is compatible (according to `compare`). The `for-yield` loop takes care of this, returning a list of TermSets which are collected together by folding the List over the union operation. As in Listing 5, we cover unexpected input cases with meaningful error messages.

3.2.5 Matching with Subterms

At this stage we have a complete implementation of abstract fingerprint indexing; as described in Schulz’s paper [Schulz 2012]. However, this index is not quite at the point where it is usable in *beagle*. Recall the main superposition rules for *beagle*’s resolution calculus (Section 2.5.3 and [Baumgartner and Waldmann 2013]). Notice in particular the condition that l may match against a *subterm* of s (this condition also exists in the original superposition calculus, Section 2.1.1). This does not match our current implementation which is only capable of indexing whole, top-level terms.

Thus our Fingerprint Index must be able to index and collect all possible matches against *subterms*. To do this we will extend *beagle*’s Term object with a function to generate all subterms; along with the position they were extracted from. For variables and constants this is trivial; we just return the symbol and Nil for the subterm

position. For functional terms however we must collect a list of each argument along with all subterms of those arguments. Listing 8 performs this operation by recursively finding the subterms of each argument and collecting them together.

```

1  /** Retrieve all subterms along with their position */
2  def subtermsWithPos : List[(Term, List[Int])] =
3      (thisterm, Nil) :: (for
4          (arg, argpos) <- args.zip(args.indices);
5          (subarg, subargpos) <- arg.subtermsWithPos
6          yield (subarg, argpos::subargpos))

```

Listing 8: Recursively grab all subterms from a complex term.

With this extension in place we can index subterms by extending our addition function (Listing 5) to also add all subterms. This process is straightforward, we simply create a new function which loops over subterms and adds each one. This will make the indexer capable of comprehensively indexing all subterms as required. Our Index however is still unfit for live use in optimising *beagle*'s inference rules. This is due to a subtle issue which prevents correct retrieval of all the information needed by the inference calculus.

3.2.6 Current Problems and Term Traces

At this stage we have a Fingerprint Index which is capable of comprehensively indexing a Clause all the way down to individual subterms. However, as previously mentioned, there is a subtle issue remaining. We shall refer to this issue as *Term Alienation*. The issue was caught during the process of *Unit Testing* (see Section 3.2.7) and relates to retrieving the Clause structure associated with any Terms retrieved as compatible.

Term Alienation

In Listing 3 we presented the actual data structure for storing indexed Terms. It presented the leaves of this index as simply a set of Term objects. At this stage we notice that this representation is actually insufficient. *Beagle*'s inference rules require knowledge of the Clauses which we are operating on, so storing only the bottom level Term does not give us enough information to perform any inferences. This is especially true considering that the stored Term may even be a subterm of what we actually wish to use for inference.

A first attempt to solve this issue involved giving every object in the expression tree (Figure 3.1) a pointer to its '*parent*' expression. However, ensuring that these pointers were correct at all times proved to be difficult. Furthermore, the pointers were difficult to use in practice since there are several steps involved in going from a subterm all the way up to a Clause; and each object must be collected along the way.

Term Tracing

Term Alienation can be better solved by introducing *Term Traces*. A Term Trace is an object which will be stored alongside any Term in our Index; containing any required information regarding where the Term originally came from. This includes pointers to each object higher up in the expression syntax tree (the associated Equation, Literal and Clause) and the (possibly nil) subterm Position. Term Traces allow us to quickly and easily retrieve any information required for inferences.

It is worth noting at this point that by indexing subterms and adding Term Traces we have increased the size and complexity of our Index data structure considerably. This is negligible however since we only introduce pointers rather than whole copies of expressions. Even in the case of copying data this would be of little concern since memory is cheap; and we are generally only concerned with speed.

3.2.7 Unit Testing with ScalaTest

As with any component of a large software project; it is vital to ensure that the Fingerprint Index functions on its own. Otherwise if we encounter any issues (after adding the Index to *beagle*) we will have no way of knowing what component is causing the problem.

Thus (in accordance with software development best practices) we must develop some unit tests which will test each individual component of our Fingerprint Index class. This is very easily done via the ScalaTest library; which integrates with the Eclipse development environment (discussed in Section 2.6) [Venners et al. 2013][Eclipse Foundation 2013].

3.3 Adding Indexing to *Beagle*

Our Fingerprint Index class is now fully capable of indexing terms for inference with the Hierarchic Superposition with Weak Abstraction Calculus. Our task now is to add the indexer itself into *beagle*'s inference loop and make use of it wherever appropriate.

Recall our analysis of *beagle*'s main loop from Section 3.1.2; where we identified the two sections most appropriate for being augmented with term indexing. We will start by adding indexing to the *superposition* inference rules; which is the primary way *beagle* and the Hierarchic Superposition with Weak Abstraction Calculus creates new information.

3.3.1 Attaching a Fingerprint Index

Actually making use of our Fingerprint Index class will require significant modification to *beagle*'s structure and proving sequence. In particular we will need to add an Index object and replace any occurrences of searching for unification matches to include indexing.

Originally indexing was included by adding a single Fingerprint Index to the main class of *beagle*. This index was initialised with all Clauses in the input knowledge set, and was given new Clauses whenever they were created. This setup caused several problems:

- **Redundant Clauses:** Some of *Beagle*'s operations (in particular Simplification, see Section 3.4.1) can cause Clauses to become *redundant*, and no longer required for inference. These Clauses would remain in the Fingerprint Index; causing clutter and unnecessary computation.
- **Difficult to Split:** The Split rule (see Section 3.1.2) could no longer be used since the Fingerprint Index could not easily be reproduced or duplicated.
- **No 'Age' Differentiation:** Recall that *beagle* maintains two collections of Clauses, `old` and `new` (see Section 3.1.2). With only one Index we currently have no way of identifying which ClauseSet an indexed Term has come from. As a result, superposition becomes unnecessarily cluttered; as it only needs to be run against Clauses from `old`.

These issues could potentially have been resolved with more careful management of the Terms in our Index, but a more elegant and simple solution exists.

Clauses from `new` are removed one by one and possibly made redundant before moving to `old`. `old` itself however is more static; it only ever has clauses *added* to it. Considering this we will add an Index to the ClauseSet class, rather than attach a single Fingerprint Index to the entire *beagle* inference process. This makes it possible to individually index each ClauseSet. By only indexing the `old` set we ensure no redundant clauses appear in inferences; and it becomes easy to only use Clauses from `old` for superposition.

Some thought is still required on how to fix the functionality of the Split rule. Split must be able to copy *beagle*'s current state to create a new parallel instance; so it must be able to copy a Fingerprint Index. When Split is activated it calls the `clone` method of the two main ClauseSets. We will have this method also copy the associated Index. We can copy an Index either by re-adding all terms to a new Fingerprint Index or by creating a deep copy of the Index pointer structure.

3.3.2 Indexing Superposition

Now that our ClauseSets are actively being Indexed we must start to make use of these Indices for superposition. As stated above superposition will only require the use of the `old` Clause Index. We are now capable of performing this retrieval; but we must still significantly modify *beagle*'s existing superposition code before we may make use of the retrieved terms.

Recall the two superposition inference rules in the Hierarchic Superposition with Weak Abstraction Calculus (taken unmodified from [Baumgartner and Waldmann

2013]):

Positive Superposition

$$\frac{l \approx r \vee C \quad s[u] \approx t \vee D}{\text{abstr}((s[r] \approx t \vee C \vee D)\sigma)}$$

Where (i) σ = simple mgu (l, u) , (ii) u is not a variable, (iii) $r\sigma \not\approx l\sigma$, (iv) $t\sigma \not\approx s\sigma$, (v) l and u are not pure background terms, (vi) $(l \approx r)\sigma$ is strictly maximal in $(l \approx r \vee C)\sigma$, and (vii) $(s \approx t)\sigma$ is strictly maximal in $(s \approx t \vee D)\sigma$

Negative Superposition

$$\frac{l \approx r \vee C \quad s[u] \not\approx t \vee D}{\text{abstr}((s[r] \not\approx t \vee C \vee D)\sigma)}$$

Where (i) σ = simple mgu (l, u) , (ii) u is not a variable, (iii) $r\sigma \not\approx l\sigma$, (iv) $t\sigma \not\approx s\sigma$, (v) l and u are not pure background terms, (vi) $(l \approx r)\sigma$ is strictly maximal in $(l \approx r \vee C)\sigma$, and (vii) $(s \not\approx t)\sigma$ is strictly maximal in $(s \not\approx t \vee D)\sigma$

When *beagle* runs these two rules it checks all Clauses in `old` against a single query Clause selected from `new`. The query clause is tested for being both the *from* clause $(l \approx r \vee C)$ or the *into* clause $(s[u] \approx t \vee D)$. Note that *beagle* does not split the superposition rules into two distinct cases; but rather generates all possible negative and positive inferences simultaneously.

Our Fingerprint Index is built to locate Terms likely to *unify*; so it is condition (i) that our Indexer is most concerned with. Condition (i) states that l and u must be unifiable by some simple most general unifier σ (refer to Sections 2.4.1 and 2.5.3 for detailed definitions of these terms). This condition implies that there are actually two distinct cases we must cover for indexing; one where l is the query term and one where u is. These cases correspond to the *from* and *into* cases mentioned above.

From Case

In this case we have a query Clause $l \approx r \vee C$ and wish to find all subterms u which are likely to unify with the top level term l . Note that we must actually attempt this for all possible l s in the Clause, so we must first loop over each *eligible* Literal. The eligible Literals are those which are positive and capable of fulfilling condition (vi) in the rules above. If an eligible Literal is unordered we must also swapping the roles of l and r . Note that Clauses maintain a list of their eligible Literals; and we do not need to search for them.

Once we have found each usable l we use our Term Index to retrieve all terms compatible for unification (see Section 3.2.4). Since Fingerprint Indexing is *non-perfect* (See Section 2.4) we must make use of *beagle*'s existing superposition code to confirm unification; and check the other inference rule conditions. Once we are sure that the superposition rule applies we may have the existing code generate the resulting

Clause.

Into Case

In this case we have a query Clause $s[u] \approx t \vee D$ and wish to find all top level terms l which are likely to unify with a subterm u . As in the from case we must find all possible terms which we can use for s while satisfying maximality and ordering conditions. After selecting a term for s we must also then loop over all its subterms, as any of these are a potential choice for u .

So for each subterm u of s we retrieve compatible Terms from the index; but unlike the from case we are not done here. l is only permitted to be a top level term so we must actually discard any subterms retrieved from the Index. This may sound expensive in terms of wasted computation; but filtering in this fashion is cheap, and avoiding this problem would require an entirely separate Index which indexes only top-level terms.

With the potential compatible values for l now retrieved *beagle's* existing code can be used to complete the inference. Our two directional cases are now covered and superposition is now using our Index to its full potential.

3.4 Extending the Fingerprint Index for Simplification

Beagle is now fully capable of indexing its terms for superposition. At this stage we could look into how the Fingerprint Index could be improved; but it is likely to be far more effective to re-examine where *beagle* now spends most of its runtime and investigate other inference procedures for which our Indexer could be applied to.

We again refer to the analysis of *beagle's* inference procedure from Section 3.1.2. In this Section we noted that there are only two subroutines of *beagle* that require searching through the ClauseSets; one being superposition and the other being simplification. So simplification is the only other area where our Indexer may be applied to any significant effect, and doing so is likely to provide a significant performance increase.

To confirm how effective Indexing simplification could be we may refer to the results when instrumenting *beagle* in VisualVM. The results in Section 4.1 indicate (as expected) that simplification often takes up a significant portion of *beagle's* runtime.

We will now attempt to apply Fingerprint Indexing to simplification; beginning by investigating the current implementation of *beagle's* simplification rules.

3.4.1 *Beagle's* Simplification Process

Beagle has several simplification rules to aid the logical inference process. These rules are not technically part of the actual rule based calculus (hence they are not mentioned in Section 2.5.3) but rather implement some special cases of those rules. Providing separate implementations of these cases can provide a significant speed-up in problems where they occur frequently.

Beagle's Clause simplification process (seen in Listing 9) does three things. First, it *demodulates* the Clause (Section 3.4.1). Second, it performs *Negative Unit Simplification* on the newly demodulated Clause (Section 3.4.1). Finally, it checks if the resulting Clause is *subsumed* by an existing one (see Section 2.4.1).

```
Simplify(select,new,old):
  posUnits := Filter (old U new) for positive, ordered unit Clauses
  negUnits := Filter (old U new) for negative unit Clauses
  simpl := Demodulate(select, posUnits)
  simpl := NegativeUnitSimplification(simpl, negUnits)
  if a Clause in (old U new) subsumes simpl
    return Empty Clause
  else
    return simpl
```

Listing 9: Pseudocode for *beagle's* Clause simplification procedure.

Negative Unit Simplification

Here we outline the process of Negative Unit Simplification, mentioned in Listing 9 above. It is used to remove Literals from Clauses which we know trivially cannot be true. Stated more precisely as an inference rule we have:

$$\text{Negative Unit Simplification} \quad \frac{l \not\approx r \quad s \approx t \vee C}{C}$$

Where there exists some *matcher* σ such that $(l \approx r)\sigma \equiv s \approx t$ (i.e. $l \approx r$ *subsumes* $s \approx t$). The clause $s \approx t \vee C$ may be removed.

This essentially says that if we know l is not equal to r , then any Literal stating otherwise may be removed. If we only looked for Literals exactly equal to $l \approx r$ however the rule would barely ever be applied and not be very useful. Thus we have Negative Unit Simplification consider any *instances* of $l \approx r$. A Literal formed by substituting the variables in $l \approx r$ also contradicts $l \not\approx r$; this is the $l \approx r$ *subsumes* $s \approx t$ condition. Checking for these subsumed literals is potentially expensive; and it is what we will attempt to improve with Fingerprint Indexing.

Note that we do not have a concept of *Positive Unit Simplification* since it would be covered as a special case of the Demodulation rule.

Demodulation

The demodulation rule was first proposed for use in the superposition calculus by Wos, Robinson, Carson, and Shalla [1967]. It allows the removal of variables and

subterms which are *redundant*; that is, their removal will not effect the truth value of any Terms they are removed from. The Demodulation rules used by *beagle* follow (recall that $l \rightarrow r$ refers to an ordered Literal $l \approx r$ and $s[u]$ refers to a Term s with a subterm u).

$$\text{Demodulation} \quad \frac{l \rightarrow r \quad s[u] \approx t \vee D}{s[r\sigma] \approx t \vee D}$$

Where σ is a matcher from l to u . (i.e. l subsumes u)

The clause $s[u] \approx t \vee D$ may be removed.

$$\text{Negative Demodulation} \quad \frac{l \rightarrow r \quad s[u] \not\approx t \vee D}{s[r\sigma] \not\approx t \vee D}$$

Where σ is a matcher from l to u . (i.e. l subsumes u)

The clause $s[u] \not\approx t \vee D$ may be removed.

At the simplest level, Demodulation allows us to replace all occurrences of l by r ; provided that we have some unit Clause $l \rightarrow r$. The rules above are more general however, and allow the replacement of any subterm u which is *subsumed* by l . As in Negative Unit Simplification, finding terms which pass this subsumption check is most time consuming. Furthermore, in Demodulation we must perform this search for all *subterms* of s ; making Demodulation a prime candidate for Indexing.

3.4.2 Generalising our Fingerprint Index

Attempting to directly apply the existing Fingerprint Index to Negative Unit Simplification and Demodulation proved difficult and produced a number of significant problems:

- **Term Matching:** The above simplification rules all try to find Terms which *match* rather than those that *unify*. Our Index so far has only been built to support unification; as that is what is used by superposition.
- **Cluttered Index:** In both Negative Unit Simplification and Demodulation we are simplifying a single Clause against a list of unit Clauses. These top-level unit Clauses are all we wish to retrieve from the Index, but our current implementation would retrieve any matching Term objects and even any matching subterms; resulting in many of the retrieved matches being thrown away.
- **Indexing Equations:** Up to this point we have only ever been interested in using a Term as the query object for our Index. Notice however that in Negative Unit Simplification we are trying to find matches for an *Equation*. In other areas

of *beagle* this issue is handled by converting the Equation into a Term using a reserved function *\$equal*:

$$\begin{aligned} l \rightarrow r &\equiv \$equal(l, r) \\ l \approx r &\equiv \$equal(l, r), \$equal(r, l) \end{aligned}$$

After conversion the *\$equal(l, r)* Term could be used as a query for our index; but since the current implementation does not convert Equations in this manner it would be guaranteed to find zero matches.

So, obviously using our current Index is not an option; and we must come up with a different method for indexing simplification. Thus, to free ourselves from the restrictions of our superposition index, we will create new Fingerprint Indices built specifically for simplification. Obviously there is no need to completely redo the existing Fingerprint Indexing code; we need only make the current implementation more flexible and configurable.

In Listing 10 we introduce an options object to pass to our Fingerprint Index class. This object will allow us to create multiple Term Indices that behave in different ways.

```

1  /** Configuration object for a Fingerprint Index */
2  class IndexConfig {
3      val positionsToSample : PositionList,
4      val indexSubterms      : Boolean,
5      val eqnToTerm          : Boolean,
6      val comparator         : (FPFeature, FPFeature) => Boolean)

```

Listing 10: Class to pass settings to an arbitrary Fingerprint Index. Note that this class does not require an implementation.

The options and their uses are outlined here:

- **positionsToSample:** A list of positions indicating what should be sampled to create term Fingerprints, with the `extractFeature` function from Listing 4.
- **indexSubterms:** Whether or not to index subterms. With this setting switched off terms are only indexed at the top level. This allows us to clear up unnecessary clutter for indexing simplification.
- **eqnToTerm:** Whether or not to convert equations to terms. In the list of problems above we pointed out that Negative Unit Simplification must convert Equations to Terms joined by *\$equal*. Thus we will require an Index which stores Equations converted in this manner.
- **comparator:** The comparison function used to compare Fingerprint Features. This function must implement a comparison table such as those seen in Section 2.4.1 or Table 3.1. Passing a different function here allows the creation of separate Indices for matching and unification.

With this options object in place we may easily create three separate Fingerprint Indices; one for Superposition, one for Negative Unit Simplification and one for Demodulation. The Simplification Indices will only ever have unit Clauses added to them meaning that we have far less to search through when trying to perform simplification. The configuration for our Indices is as follows:

- **Superposition Index:** This Index has not changed from the original implementation. It has subterm indexing on, Equation conversion off and uses unification for its comparison function (see Listing 6). Superposition requires this Index to add any Clauses added to `old` (see Section 3.1.2).
- **Negative Unit Index:** Used for indexing only the negative unit Clauses. It has subterm indexing off, Equation conversion on and uses matching for its comparison function. Negative Unit Simplification requires that this Index contains any negative unit Clauses in both `new` and `old` (see Listing 9).
- **Positive Unit Index:** Used for indexing only the positive unit Clauses. It has subterm indexing off, Equation conversion off and uses matching for its comparison function. Demodulation requires that this Index contains any positive, ordered unit Clauses in both `new` and `old` (see Listing 9).

This use of multiple indices introduces a memory overhead. However, as we have mentioned before, this overhead is negligible since we are generally only concerned with speed.

3.4.3 Applying new Indices to Simplification

Making use of these tailored Indices for simplification is now fairly trivial; as *beagle's* existing implementation does most of the work. After replacing loops over unit Clauses with loops over compatible matches from the relevant Index, the only task remaining is to ensure that the relevant unit Clauses are Indexed.

We will have the unit Clause Indices add Clauses as they are created and added to `new`. As Clauses are pulled from `new` to be added into `old` this method would seem to always contain the unit Clauses from both sets. However, notice in Listing 1 that Clauses are simplified after being selected out of `new`. This means that if simplification generates a unit Clause it will not be added into the Indices. So we must index Clauses as they added to `new` *and* as they are added to `old`.

Unfortunately this leads to many redundant Clauses in our Indices. If a unit Clause is selected from `new` there is a possibility it will simplify to the empty Clause and it will no longer be needed for inference. Otherwise it will be added into `old` and appear in the relevant index twice. These redundant and duplicate clauses could be resolved by adding the ability to *remove* from an Index. This would take significant effort however; and the extra Clauses are unlikely to cause any problems apart from a few unnecessary inferences and a negligible slowdown.

3.5 Tailoring to *Beagle's* Hierarchic Superposition with Weak Abstraction Calculus

Beagle's most time consuming search procedures now all make use of one or more Fingerprint Indices. With this fully implemented we may turn our attention toward improving lookup in the Index itself. In this section we discuss the development of some original improvements for Fingerprint Indexing, designed to specifically tailor it to *beagle's* rather unique Hierarchic Superposition with Weak Abstraction Calculus.

3.5.1 The Extended Hierarchical Unification Table

In the Hierarchic Superposition with Weak Abstraction Calculus all terms have a concept of being 'Foreground' or 'Background' (see Section 2.5); this is what makes the calculus hierarchical. Here we will attempt to use this distinction to our advantage in Fingerprint Indexing by only retrieving Terms which exist in the same hierarchy position. It is worth noting at this stage that computing where a term exists in the hierarchy is cheap (or rather, zero, as it is computed on the fly during term generation and stored for later use).

Recall the four original fingerprint feature symbols from Section 2.3. These features are computed by sampling a Term object at some position; and can be compared with a table to check if two Terms may unify or match.

- *f*: arbitrary constant function symbols.
- **A**: Variable at the exact position.
- **B**: A variable could be expanded to meet the position.
- **N**: Position can never exist regardless of variable assignment.

We introduce two new fingerprint features: **A+** and **B+**. These symbols will be used for the same purpose as the original **A** and **B**, but only for *pure background* or *abstraction* variables. These variables can only be used for pure background terms; a fact we may use to restrict the possible matches for unification.

We also wish to ensure that function symbols match in terms of being foreground or background. Since we require function symbols to be equal (see the unification and matching tables, Table 2.1 and 2.2) we would expect this to occur automatically; but this is not necessarily the case. A background function symbol may have foreground terms as arguments; in which case it will not unify with a pure background term. Thus we introduce another new Fingerprint Feature for background function symbols, *f+*. This symbol signifies a position where the entire subterm downwards from *f* is pure background. Keep in mind that this definition is slightly different to the definition for **A+** and **B+** where we only consider the variable at the precise position.

Note that by introducing these symbols we have slightly modified the definition of the original **A**, **B** and *f* features. These features will now only represent terms not

covered by the + symbols, that is, positions with foreground or impure background terms.

Table 3.1 displays the unification table with our new background feature symbols. By introducing four new symbols we have grown the table to be a considerable size. Refer to the original unification table (Table 2.1) for an in-depth explanation of how this table should be interpreted [Schulz 2012].

Table 3.1: Fingerprint comparison table for unification; extended by considering the term hierarchy.

	f_1	f_2	A	B	N	f_1+	f_2+	A+	B+
f_1	Y	N	Y	Y	N	N	N	N	N
f_2	N	Y	Y	Y	N	N	N	N	N
A	Y	Y	Y	Y	N	Y	Y	Y	Y
B	Y	Y	Y	Y	Y	Y	Y	Y	Y
N	N	N	N	Y	Y	N	N	N	Y
f_1+	N	N	Y	Y	N	Y	N	Y	Y
f_2+	N	N	Y	Y	N	N	Y	Y	Y
A+	N	N	Y	Y	N	Y	Y	Y	Y
B+	N	N	Y	Y	Y	Y	Y	Y	Y

Note that as this table is for unification it is symmetric along the leading diagonal (as in the original unification table); so we need only discuss the lower triangle of the matrix. Furthermore, notice that the bottom right segment of the table is actually identical to the original unification table. This is expected as when we compare two pure background features the comparison behaves normally.

So we need only justify the lower-left quadrant of the table. We will justify this new section line by line:

- **Pure Background Function Symbols (f_+):** Recall that this feature is only applicable if the entire subterm below f is pure background. Therefore it does not match the foreground version of the same symbol. It does however match both **A** and **B**. This is required since these symbols still match *impure* or *ordinary* background variables; which may be expanded to either foreground or pure background terms.
- **Abstraction Variables (**A+**):** Similarly to the pure background function symbol feature, this feature cannot match any terms which sit in the foreground. It can however match both **A** and **B** as they may represent either foreground or background expressions.
- **Potential Expansion of an Abstraction Variable (**B+**):** Same as for **A+** but can also match **N** (as we may choose not to generate the position).

To go with this table we present its corresponding Scala code in Listing 11. To accommodate our new Fingerprint Features we will convert our original FPA and

FPB (see Section 3.2.1) objects to take a boolean argument. We then represent **A+** and **B+** as `FPA(true)` and `FPB(true)` respectively. Unfortunately the steep increase in table size results in the amount of code required exploding. This is primarily due to that fact that we can no longer reduce the table to four simple cases as in Section 3.2.4. In that Listing we also used `Set` objects so that we did not need to provide a separate check for each direction. We can no longer use that trick since our Fingerprint Features are now parameterised instances rather than static objects.

```

1  /** Check two Fingerprint features for compatibility based
2    * on the *extended* unification table (See table in report).*/
3  def compareFeaturesForUnification
4    (a:FPFeature, b:FPFeature) : Boolean =
5    (a,b) match {
6      case (FPF(f1), FPF(f2))    => (f1.op == f2.op) &&
7                                   (if (f1.isFG || f2.isFG)
8                                     (!f1.isPureBG && !f2.isPureBG)
9                                     else true)
10     case (FPF(f), FPB(true)) => f.isPureBG
11     case (FPB(true), FPF(f)) => f.isPureBG
12     case (_, FPB(_))         => true
13     case (FPB(_), _)         => true
14     case (FPF(f), FPA(true)) => f.isPureBG
15     case (FPA(true), FPF(f)) => f.isPureBG
16     case (FPN, FPA(_))       => false
17     case (FPA(_), FPN)       => false
18     case (_, FPA(_))         => true
19     case (FPA(_), _)         => true
20     case (FPN, FPN)          => true
21     case _                   => false
22   }

```

Listing 11: Scala code implementing the Extended Hierarchical Unification Table.

3.5.2 Other Tailored Optimisations

The inference rules in the Hierarchic Superposition with Weak Abstraction Calculus carry with them *far* more restrictions than the standard superposition calculus. There are several ways which we can use these restrictions to further optimise term indexing.

Pure Background Terms

Pure Background Terms may never be a part of any superposition inference. This includes not only top-level Terms but also any pure background subterms we en-

counter. We can safely exclude any of these Terms from our superposition Term Index so long as we allow them to be included for simplification indices. To do this we will add another flag to the Fingerprint Index configuration object (see Listing 10), `IndexPureBG`, which controls the indexing of pure background Terms.

Maximal Literals

The superposition inference rules also require that the two Literals used are *strictly maximal* in their respective Clauses. *Beagle* stores these *inference eligible* maximal Literals in a list. By only indexing these Literals we save a great deal of space and time. Pure background Literals are also considered ineligible. This suits our needs for superposition but means that we must index ineligible Literals for simplification. As with the pure background Terms optimisation we implement a new configuration flag `IndexIneligible`.

Results

Chapter 3 has provided a complete account of how Fingerprint Indexing was added to *beagle*, along with some original improvements designed to tailor it to the Hierarchic Superposition with Weak Abstraction Calculus. Our task now is to determine how effective Fingerprint Indexing has been at improving *beagle*'s proving process. This chapter covers the process of testing the performance of various versions of *beagle*. This includes determining how best to define and measure performance, comparisons between indexed and unindexed *beagle* and comparisons between different Fingerprint Index configurations.

We will also provide some results taken before Fingerprint Indexing had been implemented; which were used in conjunction with our static analysis from Section 3.1.2 to identify weak points which could be improved with Term Indexing.

4.1 Instrumenting Beagle

We have stated several times throughout this report that the key area of improvement for *beagle* is in locating Terms to apply inference. It was determined through static analysis of the main loop that the two areas which performed this sort of search were superposition and simplification (see Section 3.1.2). However, we have yet to provide hard evidence that these two components result in the most significant portion of *beagle*'s runtime. This section contains some results from initial investigations (from before Fingerprint Indexing was implemented) which were used to confirm our suspicions regarding the superposition and simplification performance bottlenecks.

These investigations were performed using *VisualVM*, a tool for Java virtual machine instrumentation which is capable of automatically profiling code; that is, identifying which functions require the most computation time [Oracle Corporation 2013]. This tool can be used for Scala projects because compiled Scala code runs on the Java virtual machine; but the results can be difficult to interpret. This is due to the fact that Scala creates additional functions and renames existing ones when translating into Java. Figure 4.1 shows some typical results when instrumenting *beagle* over a single logic problem.















Hot Spots - Method	Self time [%] ▼	Self time
beagle.fol.Term\$\$anonfun\$matchers\$1.liftedTree\$1\$1 ()		2,032 ms (47.3%)
beagle.fol.unification\$.matchToList\$1 ()		1,101 ms (25.6%)
scala.collection.mutable.PriorityQueue.isEmpty ()		110 ms (2.6%)
scala.collection.immutable.VectorPointer\$class.goToNextBlockStartWritable ()		109 ms (2.5%)
scala.collection.immutable.HashMap\$.scala\$collection\$immutable\$HashMap\$\$ma		104 ms (2.4%)
scala.collection.generic.Growable\$class.\$plus\$plus\$eq ()		94.7 ms (2.2%)
beagle.util.DefaultPrinter\$.varToString ()		93.7 ms (2.2%)
beagle.calculus.derivationrules\$\$anonfun\$inPara\$2.apply ()		93.6 ms (2.2%)
scala.math.Ordering\$Int\$.compare ()		93.5 ms (2.2%)
scala.collection.TraversableLike\$class.filterNot ()		93.5 ms (2.2%)
scala.collection.MapLike\$class.isEmpty ()		93.3 ms (2.2%)
scala.collection.Iterator\$class.foreach ()		92.8 ms (2.2%)
beagle.bgtheory.SimpRule.apply ()		92.1 ms (2.1%)
scala.collection.generic.Growable\$\$anonfun\$\$plus\$plus\$eq\$1.apply ()		88.4 ms (2.1%)

Figure 4.1: Typical results when instrumenting *beagle* in VisualVM.

In the screenshot we can see there are two functions which together take up about 74% of the total runtime. These two functions consistently took up most of *beagle*'s computation time when VisualVM instrumentation was performed for a wide range of problems. For readability the function names are re-stated here:

beagle.fol.Term\$\$anonfun\$matchers\$1.liftedTree\$1\$1()

beagle.fol.unification\$.matchToList\$1()

Knowing that these two functions take the most computation time does not immediately help us with knowing the bottleneck which we must improve. To figure that out we must trace the two functions back to where they are called from. We can see from the names above that the first function is part of the Term Scala file (where Term objects and their operations are defined) and the second is from the unification Scala file (a collection of tools for performing unification and subsumption checks). Both functions are used during the process of attempting to match or unify Term objects, a process which is in turn only used during Superposition and Simplification. So we have confirmed that *beagle*'s main bottleneck is part of these two operations; specifically when they attempt to unify or match a great number of Terms. Thus by introducing Term Indexing (which drastically reduces the number of Terms we attempt inference on) we will be directly reducing strain on *beagle*'s most significant performance bottleneck.

4.2 Problem Selection

In order to test the performance of the completed implementation of Fingerprint Indexing we will make use of problems from TPTP, the Thousands of Problems for Theorem Provers library [Sutcliffe and Suttner 2013]. We will wish to compare many different versions of *beagle*, making a full run of TPTP for each version exceedingly

impractical; especially considering the available time and resources (Schulz [2012] made use of the University of Miami *Peagsus Cluster* to run his tests). Thus we must select a subset of problems which we may perform our tests against. This subset will consist of 50 TPTP problems across four problem categories.

- 10 **Arithmetic** (ARI) Problems. These problems are extremely relevant to *beagle*, as they allow it to make significant use of the integer arithmetic background reasoning module. Unfortunately this makes ARI problems too easy for *beagle* and most of them are solvable instantly without any foreground inferences. This makes them completely irrelevant to indexing; hence we only include a selection of the 10 most complex ARI problems.
- 20 **Data** (DAT) Problems. Data problems are generally quite large and require a large number of logical inferences to perform array manipulation operations. This makes them perfect for testing Term Indexing.
- 10 **Puzzle** (PUZ) Problems. Logic puzzles which are usually quite complex but occasionally can be solved instantly.
- 10 **Group Theory** (GRP) Problems. Usually quite complex but occasionally boil down to modular arithmetic problems which are solved instantly by the background prover.

In selecting the problems above we must be *fair* in the sense of not selecting problems which favour indexing; but we also are not interested in problems where indexing is not used. To achieve this we will select problems sequentially wherever possible, which prevents us from picking out specific problems where indexing performs exceptionally. Unfortunately in the case of the PUZ and GRP some problems had to be skipped over due to being slight modifications of the previous problem, instantly solvable by all versions or (in a few cases) not solvable by any version of *beagle*. For a full list of all the problems being tested against refer to Appendix A.

4.3 Indexing Metrics

Before we may begin our test runs we must decide precisely what we are testing for, that is, what we are using to measure *beagle*'s performance. In this section we propose 3 major performance indicators, total run time, *false positives* (for indexing) and run time *per inference*.

4.3.1 Total Run Time

The most obvious performance metric for any program is run time or speed. In order to measure time in our final test runs we will no longer be using VisualVM as in Section 4.1. This program is well suited to our earlier task of finding bottlenecks; but it will cause problems when attempting to analyse performance of a completed

program. In particular, attaching VisualVM to the Scala process will introduce significant overhead; introducing variance and inaccuracy in our results. Furthermore, as we have seen in Section 4.1, it can be difficult to determine how long is spent in each high-level inference rule when we only see the computation time for bottom-level helper functions.

So, as an alternative to VisualVM we will instead perform all our timing manually by injecting stopwatches into *beagle*'s code. This will allow us to directly time each of the inference rules at the top level. For the purpose of our analysis we will be timing 6 sections of *beagle*:

- **Indexing:** Total time spent adding Terms to any of the Fingerprint Indices.
- **Retrieving:** Total time spent retrieving Terms from any of the Fingerprint Indices.
- **Superposition:** Total time spent performing Superposition (includes time spent querying the index)
- **Demodulation:** Total time spent performing Demodulation (includes time spent querying the index)
- **Negative Unit Simplification:** Total time spent performing Negative Unit Simplification (includes time spent querying the index)
- **Total:** Total time for the whole proof process. This *does not* include set-up time such as parsing and interpreting the input file and initial memory allocation.

At this point it is important to discuss the effectiveness of comparing totalled timing results. While being the most intuitive measure of performance it can be misleading; since (essentially through chance) different versions of *beagle* will attempt logical inferences in different orders. This can result in a version 'getting lucky' and solving a problem instantly where others get stuck. Thus in our results we will also include a count for how many times each inference rule was used; which we may then take into consideration when comparing results.

4.3.2 False Positives

A *false positive* refers to a Term retrieved from a Fingerprint Index which does not result in an applicable inference. Each false positive results in wasted computation; making them a decent indicator of an Index's overall usefulness when solving a problem. That being said, we must take a great deal of care in interpreting the false positive count, as a low count does not necessarily indicate good overall performance. By naïvely boosting the size of the position sampling set it is possible to drop the false positive count arbitrarily low; but this will result in an overcomplicated Index which takes a long time to retrieve compatible Terms. Thus rather than strictly desiring a low false positive count our goal for best performance is to balance the count with Fingerprint length.

In some initial testing, *beagle* with indexing exhibited extremely high false positive counts. After an email correspondence with Stephan Schulz [2012] these counts were determined too high for Fingerprint Indexing to possibly be working correctly. After some analysis it was determined that many thousands of Terms retrieved from the index successfully unified but failed to perform an inference due to the large number of extra restrictions the Hierarchic Superposition with Weak Abstraction Calculus places on superposition (see Section 2.5.3). Thus the definition of a false positive was slightly modified in *beagle* to be a retrieved Term which does not successfully *unify*, no longer including any Terms which are cheaply discarded due to conditions of the calculus. This allows *beagle*'s false positive count to be more comparable to other indexing techniques based on the traditional superposition calculus. Our test statistics will also include a count for the total number of terms retrieved from Fingerprint Indices, from which subtracting the number of inferences will give the original given definition of a false positive.

4.3.3 Runtime Per Inference

In Section 4.3.1 above we mentioned that total runtime can be misleading due to ordering flukes; where a version of *beagle* may perform better since it arbitrarily performed a key inference early on. Thus a better performance indicator is time spent *per inference*, where we take the number of inferences required into account. This metric gives us a direct and clear indication of performance which (unlike the other metrics above) cannot be misinterpreted. Run time per inference is the most relevant performance indicator and the core of what this project has been trying to improve with Term Indexing.

Note that calculating the run time per inference does not require any additional code in *beagle*. Since we have total times and counts for each inference rule the runtime per inference can be calculated after each test.

4.4 Comparing Versions of *Beagle*

In this section we present results and analysis for performance against the TPTP problems and metrics discussed in Section 4.3 above. We will perform an in-depth comparison of three different versions of the prover:

- **Unmodified:** *beagle* at the commencement of this project; without any use of Fingerprint Indexing.
- **Standard:** *beagle* with the standard implementation of Fingerprint Indexing; lacking our tailored optimisations from Section 3.5. Both Superposition and Simplification are indexed.
- **Enhanced:** The full implementation described in this report; including tailored optimisations.

Note that the performance of the two indexed versions can vary depending on what positions we sample in order to generate Term Fingerprints. For the purpose of this test we will have both the standard and the enhanced versions sample the following positions:

$$\epsilon, 1, 2, 3, 1.1, 1.2$$

This set is named FP6M by Schulz [2012], and it achieved the best performance in his paper. The following Tables (4.1 and 4.2) provide the totalled performance results for the three versions over the set of 50 selected problems (See Section 4.2). We will discuss these results as a whole and then proceed to analyse results for specific problems; available in Appendix A.

Table 4.1: Totalled inference counts and indexing statistics for various versions of *beagle*.

Version	Inference Counts			Indexing Results		
	Sup	Demod	NegUnit	TotalFound	SupFP	SimpFP
Unmodified ¹	414216	29097	1826	0	0	0
Standard	162881	41414	2452	61884768	15525	39778148
Enhanced	146861	35326	1960	58119897	17641	39916687

Table 4.2: Totalled timing results for various versions of *beagle*.

Version	Time Spent (seconds)					
	Indexing	Retrieving	Sup	Demod	NegUnit	Total
Unmodified ¹	0	0	730.44	9.44	31.99	5623.21
Standard	28.4	38.73	254.17	41.66	3.18	381.36
Enhanced	18.74	17.58	168.79	30.56	2.12	259.02

4.4.1 Totalled Performance Against Indexing Metrics

In this section we will discuss the results we see in the tables of total results across the problem set (Tables 4.1 and 4.2); in relation to the performance metrics we discussed in Section 4.3. Performance results for each individual problem are available in Appendix A and will be discussed below in Sections 4.4.2 and 4.4.3.

Total Time

Looking only at the final total runtime for the three programs it would appear as though using indexing has resulted in an improvement of over 95% from 5623 seconds to 259. In addition to this, the unmodified version also failed to solve two DAT

¹This version failed to solve two problems within 8 hours (28800 seconds). These results are not included in the totals. See verbose table of results (Section A.1)

problems entirely when given 8 hours (28800 seconds) to solve each one; and accounting for these problems results in at least a 99.6% improvement! Unfortunately however this improvement cannot be entirely attributed to our implementation of Fingerprint Indexing; and as mentioned in Section 4.3 we must be careful to consider all factors. In particular, notice that the unmodified version performs 414216 superposition inferences, nearly three times that of our final enhanced implementation. This sort of discrepancy is not entirely unexpected, and is essentially a fluke introduced by changing the order inferences are performed. With this in mind the 95% improvement is no longer relevant; and we must look into alternative measures of performance.

Notice however that the standard version of beagle only performs about 10% more superposition inferences than the enhanced version; making these two versions much more comparable on total time. Between these two versions we observe a total run time improvement of 122 seconds, or about 2.5 seconds per problem. Of particular interest in comparing these two versions is the difference between time spent retrieving from the index. We notice that the enhanced version spends about 50% less time performing retrievals; even though it uses a far more complicated comparison table function (compare Listing 6 for the standard version to Listing 11 for enhanced). The speed up must then come from a reduced need to traverse the Fingerprint Index due to less matching fingerprints; implying that our tailored improvements are greatly improving the efficiency of Fingerprint Indexing.

False Positives

When we examine the count for false positives it does not immediately reflect our above finding that the enhanced version has improved the efficiency of indexing. In fact we observe over 10% *more* superposition false positives in the enhanced version (and about 0.5% more simplification false positives). This seems to contradict our raw timing results; but the contradiction may be resolved by considering all factors involved.

An indexing false positive obviously results in some unnecessary computation as we have attempted unification and failed. However, even after unification succeeds we may still fail to perform superposition due to other restrictions of the inference rule. In particular there are four Literal and Term ordering restrictions which must be checked *after* verifying unification, as they require the substitution used (see Section 2.5.3). This means that even when a term is not counted as a false positive for unification it may still cause the same amount of unnecessary computation.

Thus we must consider the TotalFound column, which counts the total number of terms retrieved from the index; regardless of whether or not they result in unification or an inference. In this column we observe a superb improvement in the enhanced version of *beagle*, with it retrieving almost four million fewer terms than the standard index. This difference is direct evidence of tailored enhancements reducing the amount of index traversal necessary; and unlike the false positive count we can be sure that each term not retrieved is saving us some amount of computation. This ex-

plains how we can have 2116 more false positives and still see a significant increase in performance; but we still have not seen how these extra false positives could have occurred.

The difference is small enough that we can again consider it a fluke arising from the variation in inference order. This is very unsatisfying however since our improved unification table from Section 3.5.1 should result specifically in fewer false positives; not just fewer terms retrieved from the index. Recall that the extended table works by considering the placement of terms in the hierarchy (whether they are foreground or background). Thus the improvement will only be relevant when there are a reasonable percentage of background terms; such as in arithmetic problems. When examining only the set of ten ARI problems (see the verbatim result tables in Appendix A) we observe that the enhanced version is superior by all measures of performance. In fact we observe a total of *zero* superposition false positives versus 124 for the standard version. This result shows the true worth of our improved unification table, as it is not necessarily relevant in the case of DAT, PUZ and GRP problems. Our additional tailored improvements from Section 3.5.2 are still relevant for these problems and produce the significant improvement seen in total terms retrieved.

Time Per Inference

Now we come to what is perhaps the most relevant and telling measure of performance; *time spent per inference*. This measure is not affected by any random factors introduced by changing the order of inference and gives us a true indication of how much we have improved each inference rule. Table 4.3 presents the results for this performance measure; derived from the totals above.

Table 4.3: Total time spent per inference for various versions of *beagle*.

Version	Superposition	Demodulation	NegUnit Simplification
Unmodified	1.7ms	0.3ms	17.5ms
Standard	1.5ms	1.0ms	1.3ms
Enhanced	1.1ms	0.8ms	1.0ms

First of all we notice that the standard version of Fingerprint Indexing has improved superposition by 2 milliseconds per inference, a 12% improvement, and our enhanced version has improved it by 6 milliseconds, a 35% improvement. While this is a far call from the 95% improvement which total run time initially implied it still presents a fantastic final result. Negative Unit Simplification produces a phenomenal 94% improvement, down to 1 millisecond per simplification from 17.5 (we will examine how this improvement occurred when examining individual problem results in Section 4.4.3). Unfortunately demodulation does not produce the positive results we see in the other inference rules and in fact our indexed versions perform far worse than the original version of *beagle*.

To explain this unfortunate result we refer again to the verbatim results in Appendix A. In particular observe the results for problem PUZ037-1.p. In the results for both the standard indexing and enhanced indexing versions of *beagle* this problem takes up about 60% of the total time spent on demodulation. Despite this, not a single demodulation inference is performed; resulting in over 38 million false positives for indexing simplification (which constitutes over 95% of the total simplification false positives for both versions). This puzzle seems to present a ‘worst case scenario’ for demodulation indexing and produces results far outside the norm. Thus it is reasonable to exclude it as an outlier and consider demodulation results without this problem included.

When excluding PUZ037-1.p we observe 0.29, 0.39 and 0.31 milliseconds per demodulation for the unmodified, standard and enhanced versions respectively. Even though we no longer see the significant loss of performance from Table 4.3, our indexed versions are still performing slightly worse than original *beagle*. Realistically they are performing about the same; considering a 0.1 millisecond margin of error. Unfortunately it seems as though *beagle*’s current implementation of demodulation is not well suited to indexing. This is perhaps due to the fact that demodulation must still loop over all subterms of its query Clause; but avoiding this would require very significant modification to *beagle*’s main loop (as simplification currently only applies to the Clause selected from *new*, see Section 3.1.2 and 3.4.1). On the other hand it is possible that the issue arises simply from the specific problem set we are testing against; since in many of results we observe very few or even zero demodulation operations. More testing is required to determine if restructuring *beagle* to improve demodulation could be worthwhile.

4.4.2 Results for Small Problems

Here we shall present some results for the smaller problems in our TPTP selection. A significant portion of the problems in TPTP are fairly small scale; and this is reflected in our results where 42 out of the 50 problems take less than 10 seconds to solve in our final indexed version. It is interesting to take a closer look at these small scale problems because they are where we should expect to see indexing perform *the worst*. In small, fast problems where our knowledge base is small and few inferences are necessary we would expect indexing to be unnecessary and only introduce complexity. Figure 4.2 presents the superposition runtime of all versions for problems where this time was less than 5 seconds; which makes up 43 of our 50 problems. The results are presented in order of superposition time for unindexed *beagle*.

As expected, indexing does not achieve the 35-95% improvement which we have observed for the whole problem set. Up until about problem 33, where the unindexed time goes over 1.5 seconds, the indexed versions of *beagle* typically perform about the same or worse than the unindexed version. After this point we begin to see indexing provide a notable improvement; particularly in our enhanced version. By inserting trend lines we can see that original *beagle* and the standard indexing version of *beagle* are achieving similar performance overall; and our enhanced version begins to come

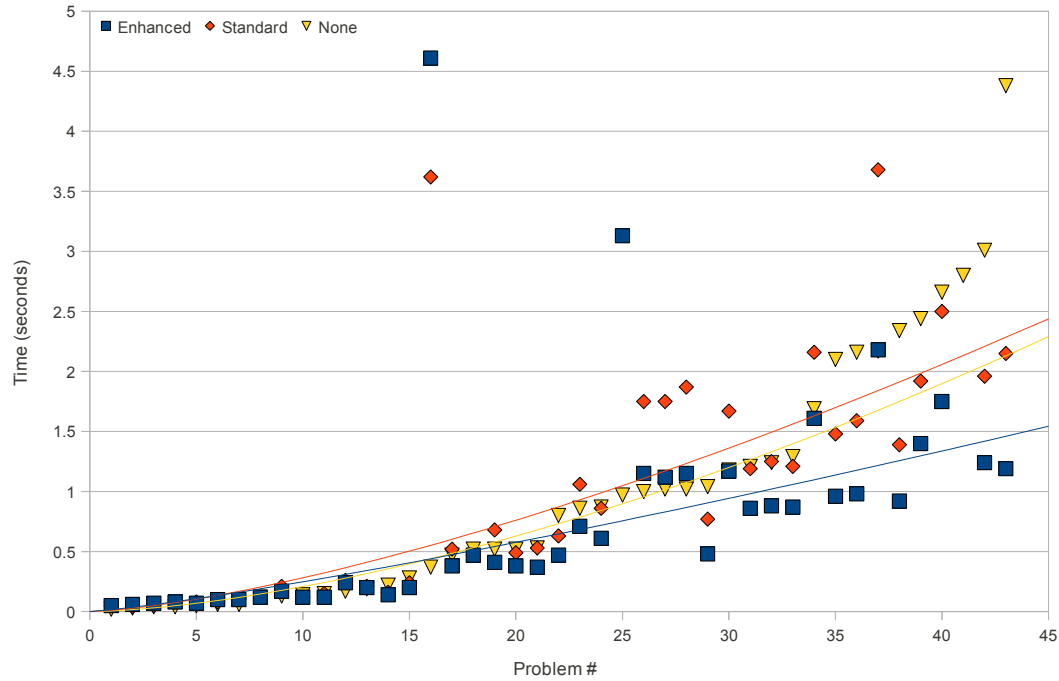


Figure 4.2: Superposition time comparison for three versions of *beagle* for small problems (under 5 seconds of superposition).

out on top after about problem 30. This shows us that even though indexing would not normally apply to these simple problems our tailored optimisations have made it worthwhile.

4.4.3 Results for Large Problems

More relevant to indexing performance are the results of large problems. When our knowledge base consists of several thousand Clauses it no longer becomes practical to search through them for inference matches and Term indexing becomes far more vital. Furthermore for each inference we perform our knowledge base grows and this problem will become worse. In this section we will discuss the results of large problems where many thousands of inferences are performed; in which we expect Fingerprint Indexing to improve performance significantly.

Most Significant Superposition Results

First of all we will examine the problems for which a significant amount of time is spent on superposition. These problems make up the 7 out of 50 problems which we did not consider in Section 4.4.2. Table 4.4 shows the superposition count and timing results for these 7 problems against all 3 versions of *beagle*; sorted in order of time spent in the unindexed version.

Table 4.4: Superposition counts and time for the 7 most extreme problem examples.

Problem	Enhanced		Standard		Unmodified	
	Sup Count	Sup Time	Sup Count	Sup Time	Sup Count	Sup Time
PUZ038-1.p	2656	24.59	2656	33.30	2579	22.6
DAT050=1.p	52438	17.53	65908	31.54	111277	48.62
DAT039=1.p	11249	13.20	11278	22.51	63897	130.77
DAT040=1.p	10551	14.49	11765	21.29	73775	190.71
DAT038=1.p	10344	12.53	11848	24.04	132216	294.86
DAT043=1.p	15422	18.67	12865	26.08	N/A ¹	N/A ¹
DAT048=1.p	14186	17.65	16462	35.77	N/A ¹	N/A ¹

Of particular note are the two final problems, which the original version of *beagle* was unable to solve and timed out after 8 hours. In our final enhanced version these two problems also present some of the largest times for superposition inferences; but in this case they are no longer than *twenty seconds*. This can be partially considered a fluke of inference ordering; but without the ability to rapidly apply inferences with indexing it would not have been possible to reach the key inferences which allowed the problem to be solved.

Most Significant Negative Unit Simplification Results

Back in Section 4.4.1 we observed a 94% improvement for each Negative Unit Simplification performed, down to 1 millisecond per simplification from 17.5. We will now look at the problems which most stress the Negative Unit Simplification rule and attempt to explain this tremendous performance increase. Looking at verbatim results from Appendix A we can observe that there are four problems from DAT for which Negative Unit Simplification takes an abnormally large portion of time. In fact, for unindexed *beagle* these four problems cover over 97% of the time spent performing Negative Unit Simplification! Table 4.5 lists the four problems along with their performance in the three versions of the prover.

¹These results are not included as the problem could not be solved within 8 hours (28800 seconds).

Table 4.5: Negative Unit Simplification counts and time for the 4 most extreme problem examples.

Problem	Enhanced		Standard		Unmodified	
	Neg Count	Neg Time	Neg Count	Neg Time	Neg Count	Neg Time
DAT039=1.p	15	0.14	15	0.2	73	4.84
DAT040=1.p	17	0.13	16	0.2	81	5.94
DAT050=1.p	1743	0.71	2252	1.33	1414	6.24
DAT038=1.p	10	0.14	10	0.19	104	14.05

The most interesting example problem here is DAT038=1.p, where we can see the unmodified version spends an enormous amount of time checking for Negative Unit Simplifications; but in the end performs comparatively few. This problem must have generated a large number of negative unit Clauses which were not necessarily applicable for Negative Unit Simplification; resulting in a great deal of unnecessary computation for unindexed *beagle*. In terms of time spent per inference, unindexed *beagle* requires 135 milliseconds for each simplification; which is nearly *eight times* its norm. It is worth noting at this point that DAT038=1.p *is not* a lone outlier; and in particular we see very similar (though not quite as extreme) results in DAT039=1.p and DAT040=1.p. These problems are also made relevant by the fact that our indexed versions are suffering from the same issue of excessive negative unit Clause generation. However, they are much more capable of coping with large quantities of negative Literals. As a result they do see worse than average simplification times here but perform far better than unindexed *beagle*.

4.5 Comparing Various Fingerprint Sampling Positions

Now that we have thoroughly explored and compared the three different versions of *beagle* it is now worth considering the impact of different Fingerprint Indexing configurations; in particular the set of positions being sampled. By increasing the number of positions sampled we should expect a decrease in false positive rate, since Terms retrieved from the Index will be compatible for unification at more points. Unfortunately however this does not necessarily correspond to an increase in overall performance. The set of positions determines how long each Term Fingerprint is and thus decides the depth of our Index tree (See Section 2.4.2). With a more complex Index structure each retrieval of compatible Terms will take more computation and more time. It is therefore important that we strike a balance between accurate comparisons and quick comparisons.

A wide variety of sampling configurations was tested by Schulz [2012], so our primary goals here are to cross-reference our results with his and confirm which set best balances Fingerprint length. For this purpose we will compare 6 of the best performing sampling sets proposed by Schulz. The sets we will be comparing are as follows: (see Section 2.4.1 for an explanation of position syntax)

- FP3W: Samples 3 positions to a maximum depth of 2. While this set creates tiny Fingerprints it was still able to achieve good results due to the relative simplicity of retrieving from the index. [Schulz 2012]

$$\epsilon, 1, 2$$

- FP4M: Samples 4 positions to a maximum depth of 3.

$$\epsilon, 1, 2, 1.1$$

- FP6M: Samples 6 positions to a maximum depth of 3. This set produced the best performance in the tests by Schulz [2012], and was used for the version comparisons in Section 4.4.

$$\epsilon, 1, 2, 3, 1.1, 1.2$$

- FP7: Samples 7 positions to a maximum depth of 3.

$$\epsilon, 1, 2, 1.1, 1.2, 2.1, 2.2$$

- FP8X2: Samples 16 positions. This is an example of excessively large Fingerprints which should result in a steep decrease in performance.

$$\epsilon, 1, 2, 3, 4, 1.1, 1.2, 1.3, 2.1, 2.2, 2.3, 3.1, 3.2, 3.3, 1.1.1, 2.1.1$$

Tables 4.6 and 4.7 provide the total performance for each Fingerprint sampling set. These totals are taken over 49 of our 50 problems. PUZ037-1.p has been excluded from these results since (as we discussed in Section 4.4.1) it presents an exceptional ‘worst case’ for demodulation indexing; making results including it difficult to compare (particularly for simplification false positives). Verbatim results for each problem along with total results over all 50 problems are available in Appendix B.

Table 4.6: Totalled inference counts and indexing statistics for various Fingerprint sampling sets.¹

Sample Set	Inference Counts			Indexing Results		
	Sup	Demod	NegUnit	TotalFound	SupFP	SimpFP
FP3W	162218	42402	2472	13913606	69429	1815992
FP4M	147798	35709	1963	13469779	26847	1851515
FP6M	144505	35326	1959	12601762	16406	1694731
FP7	160641	41435	2453	13017546	14166	1557268
FP8X2	159385	40876	2438	12819184	11229	1602033

Table 4.7: Totalled timing results for various Fingerprint sampling sets.¹

Sample Set	Time Spent (seconds)					
	Indexing	Retrieving	Sup	Demod	NegUnit	Total
FP3W	11.52	14.02	170.37	9.26	1.78	237.75
FP4M	13.09	14.12	164.95	9.51	1.82	230.68
FP6M	16.82	16.5	159.93	10.78	2.11	229.59
FP7	20.62	19.22	170.89	12.55	2.5	250.8
FP8X2	45.56	32.59	181.43	21.45	4.06	294.8

False positives are the main statistic we expect to be impacted by altering position sets. Notice that when only sampling 3 positions we see a huge number of superposition false positives, which more than halves when we move up to the FP4M set. Each extra position we sample drops the superposition false positive count further; though no other drops are as significant as the drop when moving from 3 positions to 4. The false positives for simplification are also on a falling trend; but not quite as consistently or uniformly as the false positives for superposition.

Recall that a decrease in false positives does not necessarily imply a decrease in performance. Consider the difference between time results for FP3W and FP4M. Sampling an extra position only saves us about 5 seconds of superposition time; despite the fact that we cut false positives in half. This trend continues when we add two positions to FP6M. Between FP4M and FP6M we observe a saving of 5 seconds during superposition; but after the extra time required to index and retrieve terms we only save just over a second in total runtime. After 6 positions the trend of gentle performance increase reverses; and as expected we have reached a point where increasing indexing accuracy is no longer worthwhile. On the extreme far end of the scale where we sample 16 positions (FP8X2) overall performance has dropped significantly. Even though FP8X2 has a sixth the amount of false positives that FP3W has it takes nearly a minute longer to solve all 49 problems.

These results line up with our expectations for how Fingerprint lengths impact

¹Excludes problem PUZ037-1.p

performance; and with the results from [Schulz 2012]. Like Shulz, our best result was given by FP6M. It is only the best set by a very small margin however; and both tests of smaller sets perform essentially the same if we consider a reasonable margin of error. It would appear as though the set used for Fingerprint Indexing does not have a significant impact on performance so long as Fingerprint length is kept within a range of about 3 to 6.

There is another interesting phenomenon in these results which we have not yet discussed. The total time spent on both simplification rules (Demodulation and Negative Unit Simplification) gets uniformly worse as we add sampling positions; quite significantly in the case of Demodulation. Notice that regardless of how many positions we sample we always observe a large count for simplification false positives; with only a 12% decrease from FP3W to FP8X2. This implies that the extra sampling positions are not as useful for simplification indexing, resulting in us only observing the loss of performance from complicating Term retrieval. This is possibly due to the fact that for simplification we only ever index unit Clauses; which will often be quite simple compared to the wide array of Terms seen in the superposition index. Smaller terms are less likely to be able to make use of more sampling positions, which would explain why adding them does not result in increased performance.

Conclusion

5.1 The Benefits of Term Indexing for the *Beagle* Theorem Prover

5.2 Future Improvements

Here we list some thoughts on possible improvements to *beagle* and Fingerprint indexing in general; which were either not relevant to the current work or were not investigated due to time constraints.

5.2.1 More Thorough Testing

5.2.2 Extended Data Structures

The focus of this thesis has been with high-level logical improvements. Here we present some thoughts on how 'low-level' speed could be saved at the cost of memory.

Drop the tree structure. Have an entry for each possible fingerprint and add terms to each bin they are compatible with during indexing. Moves runtime from retrieval (called VERY often) to indexing (called barely ever, comparatively) but requires a tremendous amount of memory.

5.2.3 More Fingerprint Indices

5.2.4 Extensions to Fingerprint Indexing

Symbol count / Other Features

Retrieval Caching

I believe retrieval caching could be used to significant effect for superposition indexing; in particular for the *into case* described in section 3.3.2. In this case we must loop over each subterm, meaning that we repeatedly query the index for trivial bottom-level terms like a single function symbol or variable. These cases could be cached and retrieved instantly.

Implementing this cache would require great care as any newly indexed terms must also be added to any matching cache sets. This puts a restriction on how many queries can be cached since each extra one slows down the process of adding to the index.

Dynamic Fingerprinting

Negligible due to length/FP balance. Can make static ones arbitrarily good.

Tailoring to a problem is good though, consider example where many **B**s are generated, but the position cannot be reached due to a maximum function arity.

5.2.5 Comparing Indexing Techniques

5.3 Final Thoughts

By considering more indepth results we were able to better rate the performance of Fingerprint Indexing rather than just rate it on runtime.

Result Tables for TPTP Selection

This appendix includes verbose statistics for test runs at various stages of the project. The tests are all taken from the TPTP Problem Library [Sutcliffe and Suttner 2013] and run on an Intel i5 5600k processor with 4GB of system memory.

A.0.1 Metrics

- **Inference Counts**
 - **Sup**, The number of Superposition inferences.
 - **Demod**, the number of Demodulation inferences.
 - **NegUnit**, the number of Negative Unit Simplifications.
- **Indexing Results**
 - **TotalFound**, the number of terms retrieved from the index.
 - **SupFP**, the number of 'false positives' for superposition indexing.
 - **SimpFP**, the number of 'false positives' for simplification indexing.
- **Time Spent (seconds)**
 - **Indexing, Retrieving, Sup, Demod, NegUnit, Total**.

A.0.2 Problem list

- Arithmetic Problems (ARI)

ARI601=1.p,	ARI602=1.p,	ARI603=1.p,	ARI604=1.p,	ARI605=1.p,
ARI606=1.p,	ARI607=1.p,	ARI608=1.p,	ARI609=1.p,	ARI610=1.p.
- Data Problems (DAT)

DAT031=1.p,	DAT032=1.p,	DAT033=1.p,	DAT034=1.p,	DAT035=1.p,
DAT036=1.p,	DAT037=1.p,	DAT038=1.p,	DAT039=1.p,	DAT040=1.p,
DAT041=1.p,	DAT042=1.p,	DAT043=1.p,	DAT044=1.p,	DAT045=1.p,
DAT046=1.p,	DAT047=1.p,	DAT048=1.p,	DAT049=1.p,	DAT050=1.p.
- Puzzle Problems (PUZ)

PUZ037=1.p,	PUZ038=1.p,	PUZ043=1.p,	PUZ044=1.p,	PUZ045=1.p,
PUZ047=1.p,	PUZ054=1.p,	PUZ060=1.p,	PUZ061=1.p,	PUZ062=2.p.
- Group Theory Problems (GRP)

GRP001=1.p,	GRP001=2.p,	GRP001+6.p,	GRP003=2.p,	GRP004=2.p,
GRP009=1.p,	GRP012=1.p,	GRP012=2.p,	GRP012+5.p,	GRP026=1.p.

A.1 Unmodified Beagle

Table A.1: Verbatim results for unmodified *beagle*.

Filename	Inference Counts			Indexing Results			Time Spent (seconds)					
	Sup	Demod	NegUnit	TotalFound	SupFP	SimpFP	Indexing	Retrieving	Sup	Demod	NegUnit	Total
ARI601=1.p	5	1	0	0	0	0	0	0	0.04	0.01	0	0.16
ARI602=1.p	5	1	1	0	0	0	0	0	0.03	0.01	0	0.14
ARI603=1.p	2	0	0	0	0	0	0	0	0.02	0.01	0	0.1
ARI604=1.p	10	2	0	0	0	0	0	0	0.05	0.01	0	0.19
ARI605=1.p	10	2	0	0	0	0	0	0	0.04	0.01	0	0.2
ARI606=1.p	48	5	1	0	0	0	0	0	0.17	0.03	0.01	0.48
ARI607=1.p	46	4	0	0	0	0	0	0	0.19	0.02	0.01	0.41
ARI608=1.p	30	3	1	0	0	0	0	0	0.15	0.03	0	0.41
ARI609=1.p	26	4	0	0	0	0	0	0	0.13	0.02	0	0.35
ARI610=1.p	225	60	5	0	0	0	0	0	0.52	0.09	0.02	1.68
DAT031=1.p	42	0	1	0	0	0	0	0	0.22	0.01	0.01	0.38
DAT032=1.p	750	185	25	0	0	0	0	0	1.29	0.11	0.03	2.03
DAT033=1.p	475	113	22	0	0	0	0	0	1.04	0.1	0.03	1.64
DAT034=1.p	1584	94	4	0	0	0	0	0	2.16	0.09	0.05	2.89
DAT035=1.p	2017	99	3	0	0	0	0	0	2.34	0.09	0.06	3.22
DAT036=1.p	1485	111	3	0	0	0	0	0	2.1	0.1	0.04	2.84
DAT037=1.p	220	24	2	0	0	0	0	0	0.87	0.05	0.01	1.15
DAT038=1.p	132216	1272	104	0	0	0	0	0	294.86	0.96	14.05	3835.55
DAT039=1.p	63897	1411	73	0	0	0	0	0	190.71	0.68	4.84	671.25
DAT040=1.p	73775	1613	81	0	0	0	0	0	130.77	0.69	5.94	926.53
DAT041=1.p	104	5	0	0	0	0	0	0	0.52	0.03	0.01	0.75
DAT042=1.p	1795	59	5	0	0	0	0	0	2.17	0.07	0.04	2.94
DAT043=1.p	Failed to solve after 8 hours of computation.											
DAT044=1.p	123	11	2	0	0	0	0	0	0.52	0.05	0.01	0.92
DAT045=1.p	123	11	2	0	0	0	0	0	0.53	0.05	0.01	0.93
DAT046=1.p	5595	113	5	0	0	0	0	0	4.38	0.1	0.17	8.77
DAT047=1.p	3445	113	6	0	0	0	0	0	3.01	0.09	0.11	4.66
DAT048=1.p	Failed to solve after 8 hours of computation.											
DAT049=1.p	2430	951	35	0	0	0	0	0	2.44	0.22	0.07	4.21
DAT050=1.p	111277	22295	1414	0	0	0	0	0	48.62	3.24	6.24	99.44
PUZ037-1.p	1148	0	1	0	0	0	0	0	2.8	0.88	0.01	4.17
PUZ038-1.p	2579	0	1	0	0	0	0	0	22.6	0.07	0.04	23.44
PUZ043-1.p	49	0	0	0	0	0	0	0	0.28	0.01	0	0.47
PUZ044-1.p	3	0	0	0	0	0	0	0	0.06	0.01	0	0.21
PUZ045-1.p	4	0	0	0	0	0	0	0	0.06	0.01	0	0.22
PUZ047+1.p	69	0	1	0	0	0	0	0	0.47	0.03	0	0.69
PUZ054-1.p	326	0	0	0	0	0	0	0	0.86	0.16	0.01	1.3
PUZ060+1.p	25	1	0	0	0	0	0	0	0.14	0.02	0	0.28
PUZ061+1.p	25	1	0	0	0	0	0	0	0.12	0.01	0	0.26
PUZ062-2.p	104	6	4	0	0	0	0	0	0.8	0.11	0.01	1.14
GRP001-1.p	397	31	1	0	0	0	0	0	1	0.11	0.01	1.41
GRP001-2.p	104	8	1	0	0	0	0	0	0.37	0.06	0.01	0.57
GRP001+6.p	523	2	1	0	0	0	0	0	0.97	0.08	0.01	1.32
GRP003-2.p	617	6	0	0	0	0	0	0	1.24	0.09	0.01	1.61
GRP004-2.p	585	6	0	0	0	0	0	0	1.21	0.09	0.01	1.58
GRP009-1.p	563	31	1	0	0	0	0	0	1.18	0.11	0.01	1.61
GRP012-1.p	450	47	1	0	0	0	0	0	1.02	0.1	0.01	1.42
GRP012-2.p	408	38	1	0	0	0	0	0	1.02	0.1	0.01	1.42
GRP012+5.p	1053	2	1	0	0	0	0	0	1.69	0.11	0.01	2.11
GRP026-1.p	3424	356	17	0	0	0	0	0	2.66	0.31	0.07	3.76
Totals	414216	29097	1826	0	0	0	0	0	730.44	9.44	31.99	5623.21

A.2 Beagle with Superposition and Simplification Indexing

Table A.2: Verbatim results for indexing both superposition and simplification; without any tailored enhancements.

Filename	Inference Counts			Indexing Results			Time Spent (seconds)					
	Sup	Demod	NegUnit	TotalFound	SupFP	SimpFP	Indexing	Retrieving	Sup	Demod	NegUnit	Total
ARI601=1.p	5	1	0	32	0	8	0.02	0.01	0.05	0.1	0.03	0.37
ARI602=1.p	5	1	1	30	0	2	0.02	0.01	0.04	0.02	0	0.21
ARI603=1.p	2	0	0	37	0	1	0.02	0.01	0.05	0.02	0.01	0.18
ARI604=1.p	10	2	0	56	0	11	0.03	0.02	0.08	0.02	0.01	0.28
ARI605=1.p	10	2	0	56	0	11	0.02	0.02	0.08	0.03	0	0.31
ARI606=1.p	66	13	1	800	22	366	0.26	0.09	0.26	0.09	0.01	1.22
ARI607=1.p	46	4	0	441	16	104	0.04	0.05	0.21	0.04	0.01	0.51
ARI608=1.p	26	2	1	247	10	62	0.06	0.04	0.15	0.04	0.01	0.55
ARI609=1.p	38	10	0	563	20	232	0.09	0.07	0.21	0.07	0.01	0.62
ARI610=1.p	159	61	2	1941	56	830	0.38	0.19	0.47	0.17	0.04	2.26
DAT031=1.p	25	0	0	910	0	12	0.03	0.06	0.16	0.02	0	0.36
DAT032=1.p	681	152	17	21775	0	2389	0.3	0.3	1.21	0.15	0.02	2.37
DAT033=1.p	231	54	11	6320	0	709	0.18	0.23	0.77	0.1	0.02	1.47
DAT034=1.p	640	76	2	26626	2	660	0.18	0.28	1.59	0.1	0.02	2.44
DAT035=1.p	540	55	1	21512	0	524	0.21	0.3	1.39	0.11	0.02	2.32
DAT036=1.p	607	93	0	25701	3	786	0.23	0.31	1.48	0.14	0.02	2.5
DAT037=1.p	252	20	0	7091	0	289	0.08	0.22	0.86	0.07	0.01	1.24
DAT038=1.p	11848	1018	10	792240	359	24176	0.62	2.88	24.04	0.55	0.19	28.8
DAT039=1.p	11278	1187	15	793730	367	22310	0.65	2.74	21.29	0.57	0.2	25.89
DAT040=1.p	11765	1411	16	818519	372	18893	0.62	2.78	22.51	0.63	0.2	27.18
DAT041=1.p	126	4	0	4924	2	62	0.08	0.19	0.68	0.05	0.01	1.02
DAT042=1.p	2805	128	0	133095	4	972	0.29	0.56	3.68	0.14	0.05	4.94
DAT043=1.p	12865	2089	38	926672	535	30692	0.64	3.22	26.08	0.88	0.24	31.92
DAT044=1.p	130	13	0	3659	0	176	0.17	0.17	0.49	0.07	0.01	1.07
DAT045=1.p	130	13	0	3659	0	176	0.14	0.18	0.53	0.07	0.01	1.06
DAT046=1.p	1175	175	0	47379	6	1378	0.14	0.34	2.15	0.14	0.03	2.98
DAT047=1.p	800	159	2	41208	16	1857	0.29	0.33	1.96	0.14	0.02	2.94
DAT048=1.p	16462	2358	36	1130110	426	59753	0.87	4.1	35.77	0.83	0.31	44.13
DAT049=1.p	2036	648	19	83494	0	4705	0.62	0.45	1.92	0.29	0.05	4.26
DAT050=1.p	65908	30736	2252	3442432	1520	443585	11.9	9.99	31.54	6.37	1.33	72.49
PUZ037=1.p	2356	0	1	45518135	1235	38221956	2.93	1.59	12.71	25.54	0.01	39.31
PUZ038=1.p	2656	0	0	6028228	1331	46860	2.25	1.85	33.3	0.27	0.02	34.83
PUZ043=1.p	49	0	0	839	0	0	0.04	0.05	0.24	0.03	0	0.46
PUZ044=1.p	3	0	0	121	0	0	0.03	0.02	0.08	0.02	0	0.22
PUZ045=1.p	4	0	0	179	0	0	0.03	0.03	0.1	0.02	0	0.25
PUZ047+1.p	75	0	0	9228	90	760	0.09	0.22	0.52	0.09	0.01	0.87
PUZ054=1.p	326	0	0	281126	2646	96959	0.26	0.44	1.06	0.31	0.01	1.79
PUZ060+1.p	25	1	0	158	0	14	0.03	0.03	0.12	0.02	0	0.26
PUZ061+1.p	25	1	0	169	0	14	0.03	0.03	0.12	0.02	0	0.26
PUZ062=2.p	111	5	1	44231	16	9650	0.19	0.25	0.63	0.25	0.01	1.3
GRP001=1.p	1148	52	1	76200	333	30960	0.23	0.33	1.75	0.22	0.03	2.62
GRP001=2.p	1862	133	1	229391	579	207626	0.3	0.39	3.62	0.43	0.02	4.45
GRP001+6.p	3840	0	1	530613	1889	236365	0.57	0.75	5.83	0.55	0.02	7.39
GRP003=2.p	539	6	0	43823	304	12254	0.15	0.31	1.25	0.19	0.01	1.85
GRP004=2.p	499	6	0	41390	288	11499	0.15	0.3	1.19	0.17	0.01	1.71
GRP009=1.p	1116	83	1	87199	162	40001	0.2	0.32	1.67	0.25	0.02	2.46
GRP012=1.p	1269	103	1	121649	252	61759	0.29	0.35	1.75	0.28	0.02	2.74
GRP012=2.p	1333	103	1	126868	258	62819	0.3	0.36	1.87	0.28	0.02	2.86
GRP012+5.p	1441	1	0	145772	690	56694	0.24	0.33	2.16	0.24	0.01	2.93
GRP026=1.p	3533	435	20	264190	1716	66227	0.91	0.64	2.5	0.46	0.07	4.91
Totals	162881	41414	2452	61884768	15525	39778148	28.4	38.73	254.17	41.66	3.18	381.36

A.3 Including Optimisations Tailored to the Hierarchic Superposition with Weak Abstraction Calculus

Table A.3: Verbatim results for complete Fingerprint Indexing implementation.

Filename	Inference Counts			Indexing Results			Time Spent (seconds)					
	Sup	Demod	NegUnit	TotalFound	SupFP	SimpFP	Indexing	Retrieving	Sup	Demod	NegUnit	Total
ARI601=1.p	5	1	0	18	0	8	0.03	0.01	0.07	0.02	0	0.31
ARI602=1.p	5	1	1	18	0	2	0.03	0.01	0.06	0.03	0.01	0.3
ARI603=1.p	2	0	0	9	0	1	0.03	0.01	0.05	0.02	0	0.22
ARI604=1.p	10	2	0	34	0	11	0.03	0.01	0.07	0.03	0	0.33
ARI605=1.p	10	2	0	34	0	11	0.03	0.02	0.08	0.03	0.01	0.37
ARI606=1.p	76	7	1	486	0	304	0.16	0.05	0.24	0.06	0.01	0.86
ARI607=1.p	46	4	0	255	0	104	0.05	0.03	0.2	0.04	0.01	0.54
ARI608=1.p	26	2	1	115	0	62	0.05	0.03	0.12	0.04	0.01	0.48
ARI609=1.p	38	10	0	335	0	232	0.08	0.04	0.17	0.06	0.01	0.59
ARI610=1.p	265	88	4	2629	0	1968	0.37	0.15	0.47	0.17	0.02	2.11
DAT031=1.p	25	0	0	440	0	12	0.04	0.03	0.14	0.02	0.01	0.34
DAT032=1.p	681	152	17	13017	0	2389	0.24	0.16	0.87	0.13	0.02	1.86
DAT033=1.p	231	54	11	3817	0	709	0.12	0.11	0.48	0.07	0.01	0.99
DAT034=1.p	593	52	1	13722	8	506	0.16	0.15	0.98	0.08	0.02	1.68
DAT035=1.p	491	49	1	10919	2	459	0.16	0.15	0.92	0.08	0.02	1.63
DAT036=1.p	607	93	0	14494	9	786	0.17	0.15	0.96	0.09	0.02	1.77
DAT037=1.p	252	20	0	3929	2	289	0.07	0.1	0.61	0.06	0.01	0.98
DAT038=1.p	10344	894	10	435404	440	21672	0.42	0.96	12.53	0.36	0.14	15.62
DAT039=1.p	11249	1148	15	452978	463	21853	0.42	0.99	14.49	0.4	0.14	17.72
DAT040=1.p	10551	1311	17	419729	458	17852	0.37	0.94	13.2	0.41	0.13	16.35
DAT041=1.p	126	4	0	2612	2	62	0.06	0.08	0.41	0.04	0.01	0.7
DAT042=1.p	2566	105	0	70109	14	772	0.22	0.22	2.18	0.1	0.03	3.14
DAT043=1.p	15422	2558	52	641969	775	38381	0.56	1.6	26.65	0.75	0.21	32.17
DAT044=1.p	130	13	0	1889	0	176	0.13	0.09	0.38	0.06	0.01	0.91
DAT045=1.p	130	13	0	1889	0	176	0.13	0.08	0.37	0.06	0.01	0.9
DAT046=1.p	759	116	0	19383	14	932	0.13	0.17	1.19	0.09	0.02	1.84
DAT047=1.p	840	167	4	26033	27	1963	0.23	0.18	1.24	0.09	0.02	2.01
DAT048=1.p	14186	2078	35	575621	506	45308	0.62	1.23	17.65	0.49	0.2	22.35
DAT049=1.p	1955	638	18	41931	12	4539	0.43	0.25	1.4	0.21	0.03	3.17
DAT050=1.p	52438	24770	1743	1637790	1463	363971	6.69	4.22	17.53	3.46	0.71	41.38
PUZ037-1.p	2356	0	1	45518135	1235	38221956	1.92	1.08	8.86	19.78	0.01	29.43
PUZ038-1.p	2656	0	0	6029975	1345	46860	1.53	1.37	24.59	0.2	0.02	25.75
PUZ043-1.p	49	0	0	839	0	0	0.04	0.03	0.2	0.02	0.01	0.42
PUZ044-1.p	3	0	0	121	0	0	0.04	0.03	0.1	0.03	0.01	0.28
PUZ045-1.p	4	0	0	179	0	0	0.04	0.02	0.1	0.02	0	0.29
PUZ047+1.p	75	0	0	9228	90	760	0.08	0.1	0.38	0.06	0.01	0.68
PUZ054-1.p	326	0	0	281664	3178	96959	0.2	0.22	0.71	0.24	0.01	1.31
PUZ060+1.p	25	1	0	158	0	14	0.04	0.02	0.12	0.02	0	0.3
PUZ061+1.p	25	1	0	169	0	14	0.04	0.02	0.12	0.02	0	0.29
PUZ062-2.p	111	5	1	44231	16	9650	0.16	0.16	0.47	0.19	0.01	1.04
GRP001-1.p	1148	52	1	76200	333	30960	0.17	0.17	1.15	0.17	0.02	1.81
GRP001-2.p	3161	219	1	510326	1132	470561	0.27	0.33	4.61	0.58	0.02	5.61
GRP001+6.p	3224	0	1	397601	2004	168652	0.39	0.35	3.13	0.31	0.02	4.17
GRP003-2.p	570	6	0	43631	375	11388	0.12	0.15	0.88	0.12	0.01	1.34
GRP004-2.p	536	6	0	42831	357	11242	0.12	0.15	0.86	0.12	0.01	1.31
GRP009-1.p	1116	83	1	87199	162	40001	0.13	0.19	1.17	0.18	0.02	1.81
GRP012-1.p	1064	83	1	98751	238	46272	0.22	0.19	1.12	0.19	0.02	1.87
GRP012-2.p	1124	83	1	103495	247	47076	0.22	0.19	1.15	0.19	0.02	1.9
GRP012+5.p	1696	0	1	219366	1018	122585	0.2	0.22	1.61	0.24	0.01	2.31
GRP026-1.p	3533	435	20	264190	1716	66227	0.58	0.37	1.75	0.33	0.04	3.48
Totals	146861	35326	1960	58119897	17641	39916687	18.74	17.58	168.79	30.56	2.12	259.02

Results when Varying Fingerprint Sample Positions

This appendix provides verbatim results for test runs across a variety of fingerprint ‘sampling sets’. A sampling set is a set of positions indicating where terms should be sampled in order to generate a fingerprint.

Varying the position sampling set can have a significant impact on the performance of Fingerprint Indexing. Refer to Sections 2.4.4 and 4.5 for explanations of how this occurs and how to achieve optimal performance.

The various position sampling sets in use are as follows:

- FP3W: ϵ , 1, 2
- FP4M: ϵ , 1, 2, 1.1
- FP6M: ϵ , 1, 2, 3, 1.1, 1.2
- FP7: ϵ , 1, 2, 1.1, 1.2, 2.1, 2.2
- FP8X2: ϵ , 1, 2, 3, 4, 1.1, 1.2, 1.3, 2.1, 2.2, 2.3, 3.1, 3.2, 3.3, 1.1.1, 2.1.1

The TPTP problems and metrics used are identical to the list provided in Appendix A.

B.1 Sample 1: FP3W

Table B.1: Verbatim results for complete Fingerprint Indexing implementation when sampling the FP3W set.

Filename	Inference Counts			Indexing Results			Time Spent (seconds)					
	Sup	Demod	NegUnit	TotalFound	SupFP	SimpFP	Indexing	Retrieving	Sup	Demod	NegUnit	Total
ARI601=1.p	5	1	0	18	0	8	0.03	0.01	0.07	0.02	0	0.31
ARI602=1.p	5	1	1	18	0	2	0.03	0.01	0.06	0.03	0	0.31
ARI603=1.p	2	0	0	9	0	1	0.03	0.01	0.05	0.02	0	0.22
ARI604=1.p	10	2	0	34	0	11	0.03	0.01	0.07	0.03	0	0.33
ARI605=1.p	10	2	0	34	0	11	0.03	0.01	0.07	0.03	0	0.38
ARI606=1.p	76	7	1	486	0	304	0.12	0.04	0.24	0.05	0.01	0.83
ARI607=1.p	46	4	0	255	0	104	0.04	0.02	0.2	0.03	0.01	0.52
ARI608=1.p	26	2	1	115	0	62	0.04	0.02	0.11	0.03	0.01	0.45
ARI609=1.p	38	10	0	335	0	232	0.06	0.03	0.17	0.05	0.01	0.57
ARI610=1.p	265	88	4	2629	0	1968	0.29	0.1	0.43	0.13	0.02	1.9
DAT031=1.p	25	0	0	440	0	12	0.04	0.02	0.13	0.02	0.01	0.33
DAT032=1.p	648	162	15	12368	79	2311	0.19	0.13	0.85	0.11	0.02	1.77
DAT033=1.p	231	54	11	3838	18	709	0.11	0.09	0.48	0.07	0.02	1.06
DAT034=1.p	534	56	1	12703	63	542	0.14	0.11	0.92	0.07	0.01	1.6
DAT035=1.p	566	61	1	13676	64	554	0.14	0.12	0.94	0.08	0.02	1.66
DAT036=1.p	514	75	0	11219	103	628	0.14	0.11	0.88	0.07	0.01	1.6
DAT037=1.p	252	20	0	3951	21	289	0.06	0.07	0.59	0.05	0.01	0.92
DAT038=1.p	11180	976	10	455465	2902	23621	0.33	0.85	14.3	0.3	0.11	17.35
DAT039=1.p	11417	1231	15	462451	3033	23021	0.32	0.85	15.24	0.32	0.11	18.37
DAT040=1.p	11406	1404	16	469004	3200	18817	0.32	0.89	14.33	0.34	0.11	17.56
DAT041=1.p	126	4	0	2631	10	62	0.05	0.05	0.39	0.04	0.01	0.67
DAT042=1.p	2438	101	0	72898	130	790	0.2	0.18	2.15	0.09	0.03	3.07
DAT043=1.p	15455	2589	52	640765	5424	38756	0.39	1.32	26.66	0.55	0.16	31.74
DAT044=1.p	130	13	0	1918	19	176	0.09	0.06	0.36	0.05	0.01	0.81
DAT045=1.p	130	13	0	1918	19	176	0.1	0.06	0.36	0.05	0.01	0.85
DAT046=1.p	931	125	0	23512	134	1004	0.1	0.13	1.23	0.07	0.01	1.77
DAT047=1.p	807	126	2	22710	94	1430	0.19	0.13	1.17	0.08	0.02	1.89
DAT048=1.p	14879	2563	34	632331	3737	61713	0.39	1.13	20.02	0.43	0.18	24.96
DAT049=1.p	2060	671	18	43204	289	4767	0.33	0.18	1.42	0.18	0.03	3.08
DAT050=1.p	67241	31070	2260	2182400	27591	453071	3.8	3.42	22.05	2.94	0.66	45.4
PUZ037-1.p	2356	0	1	45519539	2622	38221956	0.96	0.84	7.45	19.72	0.01	27.9
PUZ038-1.p	2656	0	0	6461273	1348	46860	0.88	1.41	24.78	0.16	0.02	25.92
PUZ043-1.p	49	0	0	839	0	0	0.04	0.02	0.19	0.02	0	0.4
PUZ044-1.p	3	0	0	121	0	0	0.03	0.02	0.09	0.02	0	0.27
PUZ045-1.p	4	0	0	179	0	0	0.04	0.02	0.1	0.02	0	0.29
PUZ047+1.p	75	0	0	9292	154	748	0.08	0.08	0.38	0.05	0.01	0.68
PUZ054-1.p	326	0	0	347557	3183	96959	0.17	0.19	0.68	0.22	0.01	1.26
PUZ060+1.p	25	1	0	158	0	14	0.04	0.01	0.12	0.02	0	0.29
PUZ061+1.p	25	1	0	169	0	14	0.04	0.02	0.12	0.02	0	0.29
PUZ062-2.p	111	5	1	44281	46	9650	0.13	0.11	0.45	0.17	0.01	0.98
GRP001-1.p	1194	60	1	83173	591	34529	0.17	0.15	1.19	0.15	0.01	1.83
GRP001-2.p	2346	194	1	381633	847	352216	0.2	0.25	3.31	0.45	0.01	4.13
GRP001+6.p	3990	0	1	567876	5779	276080	0.31	0.39	4.2	0.4	0.01	5.31
GRP003-2.p	687	6	0	81704	925	25863	0.12	0.13	0.96	0.12	0.01	1.43
GRP004-2.p	557	6	0	56633	748	15173	0.11	0.13	0.88	0.11	0.01	1.32
GRP009-1.p	816	54	1	56710	316	23585	0.1	0.15	0.99	0.15	0.01	1.56
GRP012-1.p	1043	89	1	86590	389	42117	0.16	0.16	1.14	0.17	0.01	1.79
GRP012-2.p	1520	112	1	159729	816	82570	0.21	0.18	1.42	0.21	0.01	2.2
GRP012+5.p	1657	0	1	216201	2486	105347	0.17	0.19	1.68	0.2	0.01	2.32
GRP026-1.p	3681	443	22	286153	4871	69115	0.39	0.25	1.75	0.27	0.03	3.2
Totals	164574	42402	2473	59433145	72051	40037948	12.48	14.86	177.82	28.98	1.79	265.65

B.2 Sample 2: FP4M

Table B.2: Verbatim results for complete Fingerprint Indexing implementation when sampling the FP4M set.

Filename	Inference Counts			Indexing Results			Time Spent (seconds)					
	Sup	Demod	NegUnit	TotalFound	SupFP	SimpFP	Indexing	Retrieving	Sup	Demod	NegUnit	Total
ARI601=1.p	5	1	0	18	0	8	0.03	0.01	0.07	0.02	0	0.32
ARI602=1.p	5	1	1	18	0	2	0.03	0.01	0.06	0.03	0.01	0.32
ARI603=1.p	2	0	0	9	0	1	0.03	0.01	0.05	0.02	0	0.22
ARI604=1.p	10	2	0	34	0	11	0.03	0.01	0.07	0.03	0.01	0.32
ARI605=1.p	10	2	0	34	0	11	0.03	0.01	0.07	0.03	0.01	0.35
ARI606=1.p	76	7	1	486	0	304	0.13	0.04	0.23	0.05	0.01	0.86
ARI607=1.p	46	4	0	255	0	104	0.05	0.02	0.17	0.03	0.01	0.48
ARI608=1.p	26	2	1	115	0	62	0.05	0.02	0.11	0.03	0.01	0.46
ARI609=1.p	38	10	0	335	0	232	0.07	0.03	0.17	0.06	0.01	0.57
ARI610=1.p	265	88	4	2629	0	1968	0.33	0.12	0.44	0.15	0.02	2
DAT031=1.p	25	0	0	440	0	12	0.04	0.03	0.13	0.02	0.01	0.33
DAT032=1.p	681	152	17	13027	10	2389	0.22	0.15	0.86	0.13	0.02	1.84
DAT033=1.p	231	54	11	3821	4	709	0.12	0.1	0.49	0.08	0.02	1.07
DAT034=1.p	593	52	1	13722	8	506	0.15	0.13	0.98	0.08	0.02	1.66
DAT035=1.p	491	49	1	10919	2	459	0.13	0.13	0.9	0.07	0.01	1.57
DAT036=1.p	607	93	0	14494	9	786	0.16	0.13	0.96	0.09	0.02	1.74
DAT037=1.p	252	20	0	3931	4	289	0.05	0.08	0.59	0.05	0.01	0.88
DAT038=1.p	11016	894	10	456373	473	21778	0.35	0.87	13.8	0.32	0.13	16.86
DAT039=1.p	11294	1161	15	455197	477	22021	0.35	0.86	13.94	0.34	0.12	16.98
DAT040=1.p	10551	1311	17	419729	458	17852	0.31	0.81	13.13	0.35	0.11	16.11
DAT041=1.p	126	4	0	2612	2	62	0.06	0.06	0.39	0.04	0.01	0.68
DAT042=1.p	2566	105	0	70109	14	772	0.2	0.18	2.11	0.09	0.03	3.05
DAT043=1.p	15422	2558	52	641969	775	38381	0.45	1.33	25.89	0.59	0.17	31.08
DAT044=1.p	130	13	0	1889	0	176	0.11	0.07	0.36	0.05	0.01	0.87
DAT045=1.p	130	13	0	1889	0	176	0.11	0.07	0.36	0.05	0.01	0.87
DAT046=1.p	759	116	0	19383	14	932	0.12	0.14	1.17	0.08	0.02	1.81
DAT047=1.p	840	167	4	26033	27	1963	0.2	0.15	1.22	0.09	0.02	1.98
DAT048=1.p	15737	2348	37	647802	545	56268	0.49	1.2	20.45	0.47	0.18	25.54
DAT049=1.p	1955	638	18	41931	12	4539	0.35	0.19	1.36	0.19	0.03	3.01
DAT050=1.p	52438	24770	1743	1637790	1463	363971	4.27	3.08	17.61	2.69	0.58	38.32
PUZ037=1.p	2356	0	1	45519539	2622	38221956	1.28	0.9	8.37	22.02	0.01	31.16
PUZ038=1.p	2656	0	0	6461273	1348	46860	1.15	1.34	25.68	0.18	0.02	26.84
PUZ043=1.p	49	0	0	839	0	0	0.04	0.02	0.2	0.02	0.01	0.41
PUZ044=1.p	3	0	0	121	0	0	0.03	0.02	0.09	0.02	0	0.26
PUZ045=1.p	4	0	0	179	0	0	0.04	0.02	0.1	0.02	0	0.29
PUZ047=1.p	75	0	0	9292	154	748	0.07	0.09	0.38	0.06	0.01	0.68
PUZ054=1.p	326	0	0	347557	3183	96959	0.19	0.21	0.71	0.23	0.01	1.29
PUZ060=1.p	25	1	0	158	0	14	0.04	0.02	0.12	0.02	0.01	0.29
PUZ061=1.p	25	1	0	169	0	14	0.04	0.02	0.12	0.02	0.01	0.29
PUZ062=2.p	111	5	1	44247	20	9650	0.14	0.14	0.45	0.18	0.01	0.99
GRP001=1.p	1194	60	1	83165	583	34529	0.18	0.16	1.19	0.16	0.01	1.85
GRP001=2.p	3161	219	1	510368	1168	470561	0.24	0.32	4.6	0.58	0.01	5.59
GRP001+6.p	3990	0	1	567876	5779	276080	0.36	0.41	4.33	0.41	0.01	5.51
GRP003=2.p	687	6	0	80894	878	25104	0.13	0.15	0.97	0.12	0.01	1.45
GRP004=2.p	557	6	0	56589	709	15173	0.12	0.13	0.88	0.11	0.01	1.32
GRP009=1.p	1058	163	1	119150	400	70583	0.25	0.18	1.13	0.2	0.01	1.94
GRP012=1.p	1043	89	1	86586	385	42117	0.18	0.16	1.11	0.17	0.01	1.77
GRP012=2.p	1169	81	1	111969	586	51917	0.21	0.17	1.21	0.18	0.01	1.96
GRP012+5.p	1657	0	1	216201	2486	105347	0.2	0.22	1.78	0.22	0.01	2.49
GRP026=1.p	3681	443	22	286153	4871	69115	0.46	0.3	1.76	0.29	0.03	3.31
Totals	150154	35709	1964	58989318	29469	40073471	14.37	15.02	173.32	31.53	1.83	261.84

B.3 Sample 3: FP6M

Table B.3: Verbatim results for complete Fingerprint Indexing implementation when sampling the FP6M set.

Filename	Inference Counts			Indexing Results			Time Spent (seconds)					
	Sup	Demod	NegUnit	TotalFound	SupFP	SimpFP	Indexing	Retrieving	Sup	Demod	NegUnit	Total
ARI601=1.p	5	1	0	18	0	8	0.03	0.01	0.07	0.02	0	0.31
ARI602=1.p	5	1	1	18	0	2	0.03	0.01	0.06	0.03	0.01	0.3
ARI603=1.p	2	0	0	9	0	1	0.03	0.01	0.05	0.02	0	0.22
ARI604=1.p	10	2	0	34	0	11	0.03	0.01	0.07	0.03	0	0.33
ARI605=1.p	10	2	0	34	0	11	0.03	0.02	0.08	0.03	0.01	0.37
ARI606=1.p	76	7	1	486	0	304	0.16	0.05	0.24	0.06	0.01	0.86
ARI607=1.p	46	4	0	255	0	104	0.05	0.03	0.2	0.04	0.01	0.54
ARI608=1.p	26	2	1	115	0	62	0.05	0.03	0.12	0.04	0.01	0.48
ARI609=1.p	38	10	0	335	0	232	0.08	0.04	0.17	0.06	0.01	0.59
ARI610=1.p	265	88	4	2629	0	1968	0.37	0.15	0.47	0.17	0.02	2.11
DAT031=1.p	25	0	0	440	0	12	0.04	0.03	0.14	0.02	0.01	0.34
DAT032=1.p	681	152	17	13017	0	2389	0.24	0.16	0.87	0.13	0.02	1.86
DAT033=1.p	231	54	11	3817	0	709	0.12	0.11	0.48	0.07	0.01	0.99
DAT034=1.p	593	52	1	13722	8	506	0.16	0.15	0.98	0.08	0.02	1.68
DAT035=1.p	491	49	1	10919	2	459	0.16	0.15	0.92	0.08	0.02	1.63
DAT036=1.p	607	93	0	14494	9	786	0.17	0.15	0.96	0.09	0.02	1.77
DAT037=1.p	252	20	0	3929	2	289	0.07	0.1	0.61	0.06	0.01	0.98
DAT038=1.p	10344	894	10	435404	440	21672	0.42	0.96	12.53	0.36	0.14	15.62
DAT039=1.p	11249	1148	15	452978	463	21853	0.42	0.99	14.49	0.4	0.14	17.72
DAT040=1.p	10551	1311	17	419729	458	17852	0.37	0.94	13.2	0.41	0.13	16.35
DAT041=1.p	126	4	0	2612	2	62	0.06	0.08	0.41	0.04	0.01	0.7
DAT042=1.p	2566	105	0	70109	14	772	0.22	0.22	2.18	0.1	0.03	3.14
DAT043=1.p	15422	2558	52	641969	775	38381	0.56	1.6	26.65	0.75	0.21	32.17
DAT044=1.p	130	13	0	1889	0	176	0.13	0.09	0.38	0.06	0.01	0.91
DAT045=1.p	130	13	0	1889	0	176	0.13	0.08	0.37	0.06	0.01	0.9
DAT046=1.p	759	116	0	19383	14	932	0.13	0.17	1.19	0.09	0.02	1.84
DAT047=1.p	840	167	4	26033	27	1963	0.23	0.18	1.24	0.09	0.02	2.01
DAT048=1.p	14186	2078	35	575621	506	45308	0.62	1.23	17.65	0.49	0.2	22.35
DAT049=1.p	1955	638	18	41931	12	4539	0.43	0.25	1.4	0.21	0.03	3.17
DAT050=1.p	52438	24770	1743	1637790	1463	363971	6.69	4.22	17.53	3.46	0.71	41.38
PUZ037-1.p	2356	0	1	45518135	1235	38221956	1.92	1.08	8.86	19.78	0.01	29.43
PUZ038-1.p	2656	0	0	6029975	1345	46860	1.53	1.37	24.59	0.2	0.02	25.75
PUZ043-1.p	49	0	0	839	0	0	0.04	0.03	0.2	0.02	0.01	0.42
PUZ044-1.p	3	0	0	121	0	0	0.04	0.03	0.1	0.03	0.01	0.28
PUZ045-1.p	4	0	0	179	0	0	0.04	0.02	0.1	0.02	0	0.29
PUZ047+1.p	75	0	0	9228	90	760	0.08	0.1	0.38	0.06	0.01	0.68
PUZ054-1.p	326	0	0	281664	3178	96959	0.2	0.22	0.71	0.24	0.01	1.31
PUZ060+1.p	25	1	0	158	0	14	0.04	0.02	0.12	0.02	0	0.3
PUZ061+1.p	25	1	0	169	0	14	0.04	0.02	0.12	0.02	0	0.29
PUZ062-2.p	111	5	1	44231	16	9650	0.16	0.16	0.47	0.19	0.01	1.04
GRP001-1.p	1148	52	1	76200	333	30960	0.17	0.17	1.15	0.17	0.02	1.81
GRP001-2.p	3161	219	1	510326	1132	470561	0.27	0.33	4.61	0.58	0.02	5.61
GRP001+6.p	3224	0	1	397601	2004	168652	0.39	0.35	3.13	0.31	0.02	4.17
GRP003-2.p	570	6	0	43631	375	11388	0.12	0.15	0.88	0.12	0.01	1.34
GRP004-2.p	536	6	0	42831	357	11242	0.12	0.15	0.86	0.12	0.01	1.31
GRP009-1.p	1116	83	1	87199	162	40001	0.13	0.19	1.17	0.18	0.02	1.81
GRP012-1.p	1064	83	1	98751	238	46272	0.22	0.19	1.12	0.19	0.02	1.87
GRP012-2.p	1124	83	1	103495	247	47076	0.22	0.19	1.15	0.19	0.02	1.9
GRP012+5.p	1696	0	1	219366	1018	122585	0.2	0.22	1.61	0.24	0.01	2.31
GRP026-1.p	3533	435	20	264190	1716	66227	0.58	0.37	1.75	0.33	0.04	3.48
Totals	146861	35326	1960	58119897	17641	39916687	18.74	17.58	168.79	30.56	2.12	259.02

B.4 Sample 4: FP7

Table B.4: Verbatim results for complete Fingerprint Indexing implementation when sampling the FP7 set.

Filename	Inference Counts			Indexing Results			Time Spent (seconds)					
	Sup	Demod	NegUnit	TotalFound	SupFP	SimpFP	Indexing	Retrieving	Sup	Demod	NegUnit	Total
ARI601=1.p	5	1	0	18	0	8	0.02	0.01	0.07	0.02	0	0.3
ARI602=1.p	5	1	1	18	0	2	0.03	0.01	0.07	0.03	0.01	0.32
ARI603=1.p	2	0	0	9	0	1	0.03	0.01	0.05	0.02	0.01	0.23
ARI604=1.p	10	2	0	34	0	11	0.04	0.02	0.08	0.03	0.01	0.35
ARI605=1.p	10	2	0	34	0	11	0.03	0.02	0.07	0.04	0.01	0.35
ARI606=1.p	76	7	1	486	0	304	0.17	0.06	0.25	0.06	0.01	0.89
ARI607=1.p	46	4	0	255	0	104	0.05	0.04	0.2	0.05	0.01	0.57
ARI608=1.p	26	2	1	115	0	62	0.06	0.03	0.14	0.04	0.01	0.56
ARI609=1.p	38	10	0	335	0	232	0.08	0.05	0.18	0.07	0.01	0.65
ARI610=1.p	265	88	4	2629	0	1968	0.4	0.16	0.47	0.18	0.02	2.15
DAT031=1.p	25	0	0	440	0	12	0.05	0.03	0.15	0.02	0.01	0.36
DAT032=1.p	681	152	17	13017	0	2389	0.26	0.17	0.88	0.13	0.02	1.92
DAT033=1.p	231	54	11	3817	0	709	0.16	0.13	0.51	0.09	0.02	1.16
DAT034=1.p	640	76	2	15209	2	660	0.18	0.17	1.04	0.09	0.02	1.83
DAT035=1.p	540	55	1	12306	0	524	0.18	0.16	0.96	0.09	0.02	1.74
DAT036=1.p	607	93	0	14486	3	786	0.19	0.16	0.99	0.1	0.02	1.83
DAT037=1.p	252	20	0	3927	0	289	0.08	0.11	0.63	0.06	0.01	1
DAT038=1.p	11848	1018	10	474801	359	24176	0.47	1.1	16.73	0.43	0.16	20.35
DAT039=1.p	11278	1187	15	467969	367	22310	0.47	1.07	14.48	0.44	0.15	17.86
DAT040=1.p	11765	1411	16	475518	372	18893	0.48	1.1	15.38	0.47	0.15	19.01
DAT041=1.p	126	4	0	2612	2	62	0.07	0.09	0.43	0.04	0.01	0.74
DAT042=1.p	2805	128	0	83105	4	972	0.24	0.25	2.32	0.12	0.04	3.34
DAT043=1.p	12865	2089	38	531396	535	30692	0.56	1.39	18.67	0.72	0.19	23.29
DAT044=1.p	130	13	0	1889	0	176	0.14	0.09	0.38	0.06	0.01	0.95
DAT045=1.p	130	13	0	1889	0	176	0.14	0.09	0.38	0.06	0.01	0.95
DAT046=1.p	1175	175	0	28190	6	1378	0.13	0.19	1.44	0.11	0.02	2.16
DAT047=1.p	800	159	2	25390	16	1857	0.24	0.17	1.19	0.1	0.02	1.99
DAT048=1.p	16462	2358	36	704208	426	59753	0.66	1.74	25.16	0.65	0.25	31.32
DAT049=1.p	2036	648	19	43897	0	4705	0.47	0.28	1.44	0.23	0.04	3.3
DAT050=1.p	65908	30736	2252	2078173	1520	443585	9.51	5.81	21.97	4.85	0.95	53.14
PUZ037=1.p	2356	0	1	45518135	1235	38221956	2.29	1.07	9.65	20.05	0.01	30.58
PUZ038=1.p	2656	0	0	6028228	1331	46860	1.8	1.44	25.15	0.22	0.02	26.4
PUZ043=1.p	49	0	0	839	0	0	0.04	0.04	0.21	0.03	0.01	0.45
PUZ044=1.p	3	0	0	121	0	0	0.04	0.03	0.09	0.03	0.01	0.28
PUZ045=1.p	4	0	0	179	0	0	0.04	0.02	0.11	0.03	0.01	0.33
PUZ047+1.p	75	0	0	9228	90	760	0.09	0.12	0.41	0.07	0.01	0.74
PUZ054=1.p	326	0	0	281126	2646	96959	0.21	0.25	0.73	0.24	0.01	1.35
PUZ060+1.p	25	1	0	158	0	14	0.05	0.02	0.13	0.02	0.01	0.32
PUZ061+1.p	25	1	0	169	0	14	0.04	0.02	0.13	0.02	0.01	0.3
PUZ062=2.p	111	5	1	44231	16	9650	0.17	0.16	0.46	0.19	0.01	1.04
GRP001=1.p	1148	52	1	76200	333	30960	0.17	0.19	1.18	0.17	0.02	1.85
GRP001=2.p	1862	133	1	229391	579	207626	0.22	0.23	2.55	0.31	0.01	3.2
GRP001+6.p	3840	0	1	530613	1889	236365	0.45	0.43	4.21	0.41	0.02	5.44
GRP003=2.p	539	6	0	43823	304	12254	0.13	0.16	0.88	0.12	0.01	1.35
GRP004=2.p	499	6	0	41390	288	11499	0.13	0.17	0.87	0.13	0.01	1.35
GRP009=1.p	1116	83	1	87199	162	40001	0.13	0.2	1.19	0.19	0.02	1.85
GRP012=1.p	1269	103	1	121649	252	61759	0.24	0.21	1.28	0.21	0.02	2.08
GRP012=2.p	1333	103	1	126868	258	62819	0.24	0.22	1.32	0.22	0.02	2.13
GRP012+5.p	1441	1	0	145772	690	56694	0.2	0.2	1.53	0.19	0.01	2.17
GRP026=1.p	3533	435	20	264190	1716	66227	0.64	0.4	1.75	0.35	0.04	3.56
Totals	162997	41435	2454	58535681	15401	39779224	22.91	20.29	180.54	32.6	2.51	281.38

B.5 Sample 5: FP8X2

Table B.5: Verbatim results for complete Fingerprint Indexing implementation when sampling the FP8X2 set.

Filename	Inference Counts			Indexing Results			Time Spent (seconds)					
	Sup	Demod	NegUnit	TotalFound	SupFP	SimpFP	Indexing	Retrieving	Sup	Demod	NegUnit	Total
ARI601=1.p	5	1	0	18	0	8	0.03	0.02	0.08	0.02	0	0.3
ARI602=1.p	5	1	1	18	0	2	0.03	0.02	0.07	0.03	0.01	0.33
ARI603=1.p	2	0	0	9	0	1	0.04	0.01	0.06	0.02	0	0.22
ARI604=1.p	10	2	0	34	0	11	0.04	0.03	0.09	0.04	0.01	0.36
ARI605=1.p	10	2	0	34	0	11	0.04	0.03	0.07	0.04	0.01	0.37
ARI606=1.p	76	7	1	486	0	304	0.22	0.08	0.23	0.07	0.01	0.91
ARI607=1.p	46	4	0	255	0	104	0.06	0.06	0.21	0.05	0.01	0.57
ARI608=1.p	26	2	1	115	0	62	0.09	0.05	0.13	0.06	0.01	0.57
ARI609=1.p	38	10	0	335	0	232	0.12	0.09	0.18	0.09	0.01	0.69
ARI610=1.p	265	88	4	2629	0	1968	0.66	0.25	0.46	0.24	0.03	2.44
DAT031=1.p	25	0	0	440	0	12	0.06	0.06	0.15	0.03	0.01	0.39
DAT032=1.p	681	152	17	13017	0	2389	0.35	0.25	0.91	0.17	0.03	2.07
DAT033=1.p	231	54	11	3817	0	709	0.19	0.17	0.54	0.11	0.02	1.21
DAT034=1.p	640	76	2	15209	2	660	0.21	0.21	1.05	0.12	0.02	1.85
DAT035=1.p	540	55	1	12306	0	524	0.21	0.18	0.97	0.1	0.02	1.77
DAT036=1.p	607	93	0	14485	3	786	0.22	0.2	0.99	0.12	0.02	1.87
DAT037=1.p	252	20	0	3927	0	289	0.1	0.15	0.66	0.07	0.01	1.03
DAT038=1.p	11848	1018	10	473744	359	24176	0.85	1.62	16.21	0.72	0.24	20.47
DAT039=1.p	11278	1187	15	466870	367	22310	0.88	1.64	14.4	0.75	0.24	18.52
DAT040=1.p	11765	1411	16	474389	372	18893	0.94	1.78	15.75	0.84	0.24	20.32
DAT041=1.p	126	4	0	2612	2	62	0.09	0.13	0.45	0.05	0.01	0.77
DAT042=1.p	2805	128	0	83098	4	972	0.35	0.37	2.51	0.18	0.06	3.75
DAT043=1.p	12865	2089	38	529936	535	30692	1.12	2.4	19.66	1.29	0.3	25.64
DAT044=1.p	130	13	0	1889	0	176	0.18	0.16	0.43	0.08	0.01	1.05
DAT045=1.p	130	13	0	1889	0	176	0.18	0.15	0.4	0.08	0.01	0.98
DAT046=1.p	1175	175	0	28190	6	1378	0.15	0.25	1.58	0.14	0.03	2.36
DAT047=1.p	800	159	2	25390	16	1857	0.36	0.23	1.29	0.13	0.02	2.23
DAT048=1.p	16462	2358	36	703014	426	59753	1.47	2.59	26.34	1.22	0.4	34.49
DAT049=1.p	2036	648	19	43897	0	4705	0.85	0.5	1.54	0.39	0.06	3.97
DAT050=1.p	64186	30240	2236	2036061	1376	430329	26.35	12.56	24.04	9.99	1.82	79.09
PUZ037=1.p	2356	0	1	45517413	525	38221956	6.17	1.64	13.75	21.3	0.01	36.21
PUZ038=1.p	2656	0	0	5853461	1299	46860	4.44	1.78	28.96	0.4	0.02	30.53
PUZ043=1.p	49	0	0	839	0	0	0.05	0.06	0.22	0.04	0.01	0.47
PUZ044=1.p	3	0	0	121	0	0	0.04	0.03	0.1	0.03	0.01	0.28
PUZ045=1.p	4	0	0	179	0	0	0.04	0.04	0.12	0.03	0.01	0.32
PUZ047=1.p	75	0	0	9175	54	748	0.11	0.17	0.41	0.09	0.01	0.77
PUZ054=1.p	326	0	0	235424	1400	96959	0.29	0.39	0.82	0.32	0.01	1.56
PUZ060=1.p	25	1	0	158	0	14	0.05	0.04	0.14	0.03	0.01	0.33
PUZ061=1.p	25	1	0	169	0	14	0.05	0.04	0.13	0.03	0.01	0.32
PUZ062=2.p	111	5	1	44213	10	9650	0.23	0.22	0.49	0.22	0.01	1.16
GRP001=1.p	1148	52	1	76200	333	30960	0.23	0.26	1.23	0.21	0.03	1.98
GRP001=2.p	2396	131	1	265160	692	237474	0.39	0.33	3.7	0.4	0.02	4.5
GRP001+6.p	3790	0	1	507943	1511	217361	0.56	0.61	4.35	0.51	0.03	5.75
GRP003=2.p	539	6	0	43817	298	12254	0.17	0.22	0.94	0.15	0.02	1.46
GRP004=2.p	499	6	0	41384	282	11499	0.16	0.2	0.88	0.14	0.02	1.37
GRP009=1.p	1116	83	1	87191	155	40001	0.16	0.27	1.21	0.23	0.02	1.93
GRP012=1.p	1141	87	1	107105	228	51659	0.31	0.3	1.24	0.26	0.03	2.13
GRP012=2.p	1202	87	1	112052	234	52567	0.32	0.32	1.34	0.27	0.03	2.26
GRP012+5.p	1872	0	1	244398	885	128090	0.28	0.36	1.86	0.32	0.02	2.7
GRP026=1.p	3343	407	20	252082	380	62372	1.24	0.71	1.84	0.53	0.07	4.39
Totals	161741	40876	2439	58336597	11754	39823989	51.73	34.23	195.18	42.75	4.07	331.01

Bibliography

- ASPERTI, A. AND TASSI, E. 2010. Smart matching. In S. AUTEXIER, J. CALMET, D. DELAHAYE, P. ION, L. RIDEAU, R. RIOBOO, AND A. SEXTON Eds., *Intelligent Computer Mathematics*, Volume 6167 of *Lecture Notes in Computer Science*, pp. 263–277. Springer Berlin Heidelberg. (p.5)
- BACHMAIR, L. AND GANZINGER, H. 1994. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.* 4, 3, 217–247. (p.5)
- BACHMAIR, L., GANZINGER, H., AND WALDMANN, U. 1994. Refutational theorem proving for hierarchic first-order theories. *Applicable Algebra in Engineering, Communication and Computing* 5, 3-4, 193–212. (pp.11, 12)
- BAUMGARTNER, P. AND WALDMANN, U. 2013. Hierarchic superposition with weak abstraction. In M. BONACINA Ed., *Automated Deduction – CADE-24*, Volume 7898 of *Lecture Notes in Computer Science*, pp. 39–57. Springer Berlin Heidelberg. (pp.10, 11, 12, 13, 24, 27)
- DRAGOS, I. 2013. Scala IDE for Eclipse. <http://scala-ide.org/>. (p.14)
- ECLIPSE FOUNDATION. 2013. Eclipse homepage. <http://www.eclipse.org/>. (pp.14, 26)
- ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE (EPFL). 2013. Scala homepage. <http://www.scala-lang.org/>. (p.14)
- GRAF, P. AND FEHRER, D. 1998. Term indexing. In W. BIBEL AND P. SCHMITT Eds., *Automated Deduction – A Basis for Applications*, Volume 9 of *Applied Logic Series*, pp. 125–147. Springer Netherlands. (pp.6, 10)
- MCCUNE, W. 1990. Experiments with discrimination-tree indexing and path indexing for term retrieval. *JOURNAL OF AUTOMATED REASONING* 9, 9–2. (p.6)
- ORACLE CORPORATION. 2013. VisualVM All-in-One Troubleshooting Tool. <http://visualvm.java.net/>. (p.39)
- RIAZANOV, A. AND VORONKOV, A. 1999. Vampire. In *Automated Deduction – CADE-16*, Volume 1632 of *Lecture Notes in Computer Science*, pp. 292–296. Springer Berlin Heidelberg. (pp.5, 7)
- SCHULZ, S. 2002. E – A Brainiac Theorem Prover. *Journal of AI Communications* 15, 2/3, 111–126. (p.5)
- SCHULZ, S. 2012. Fingerprint indexing for paramodulation and rewriting. In B. GRAMLICH, D. MILLER, AND U. SATTLER Eds., *Automated Reasoning*, Volume

7364 of *Lecture Notes in Computer Science*, pp. 477–483. Springer Berlin Heidelberg. (pp. 4, 7, 8, 9, 10, 15, 19, 20, 21, 23, 24, 35, 41, 43, 44, 51, 53)

SUTCLIFFE, G. AND SUTTNER, C. 2013. The tptp problem library for automated theorem proving. <http://www.cs.miami.edu/~tptp/>. (pp. 40, 57)

VENNERS, B., BERGER, G., AND SENG, C. 2013. Scalatest homepage. <http://www.scalatest.org/>. (pp. 14, 26)

WEIDENBACH, C., AFSHORDEL, B., BRAHM, U., COHRS, C., ENGEL, T., KEEN, E., THEOBALT, C., AND TOPIĆ, D. 1999. System description: Spass version 1.0.0. In *Automated Deduction – CADE-16*, Volume 1632 of *Lecture Notes in Computer Science*, pp. 378–382. Springer Berlin Heidelberg. (p. 5)

WOS, L., ROBINSON, G. A., CARSON, D. F., AND SHALLA, L. 1967. The concept of demodulation in theorem proving. *J. ACM* 14, 4 (Oct.), 698–709. (p. 30)