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

September 2013

Typeset in Palatino by TEX and LATEX 2ε.

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

Timothy Clarence Richard Cosgrove
29 September 2013

For Dana.

Acknowledgements

Thank you to my Supervisor and all...

Abstract

This should be the abstract to your thesis...

Contents

Acknowledgements **vii**

Abstract **ix**

1 Introduction **1**
1.1 Motivation . . . 1
1.1.1 A Theoretical Framework . . . 1

2 Background **3**
2.1 First-Order Logic Terms and Notation . . . 3
2.1.1 FOL basics . . . 3
2.1.2 FOL with Equality and Ordering . . . 3
2.1.3 Logic Terminology used in this Paper . . . 3
Positions . . . 3
Substitution . . . 4
2.1.4 Popular Problems in First Order Logic . . . 4
2.1.5 The Superposition Calculus . . . 4
2.2 Automated Reasoning and Theorem Proving . . . 5
2.2.1 SPASS . . . 5
2.2.2 Vampire . . . 5
2.2.3 E . . . 5
2.3 Term Indexing . . . 5
Top Symbol Hashing . . . 5
Discriminant Trees . . . 5
2.4 Fingerprint Indexing . . . 5
2.4.1 Term Fingerprints . . . 5
2.4.2 The Fingerprint Index . . . 7
2.4.3 Position Variants . . . 7
2.5 The *Beagle* Theorem Prover . . . 7
2.5.1 Hierarchic Reasoning . . . 7
2.5.2 Weak Abstraction . . . 7
2.5.3 Rule Based Inference System . . . 8
2.5.4 *Beagle*'s Shortcomings . . . 9
2.6 Tools Used . . . 9
2.6.1 Scala . . . 9
2.6.2 VisualVM . . . 9
2.6.3 Eclipse . . . 9

3 Implementing Fingerprint Indexing **11**
3.1 Structure of *Beagle* . . . 11
3.1.1 Syntax and Data Structures . . . 11
3.1.2 Main Inference Procedure . . . 13
3.2 Building the Fingerprint Indexer . . . 14
3.2.1 Objects and Data Types . . . 15
Positions . . . 15
Fingerprint Features . . . 15
Term Fingerprints . . . 15
Fingerprint Index . . . 15
3.2.2 Building Term Fingerprints . . . 16
3.2.3 Adding Terms to the Index . . . 17
3.2.4 Retrieving Compatible terms . . . 18
3.2.5 Matching with Subterms . . . 20
3.2.6 Current Problems with the Fingerprint Index . . . 21
Data loss . . . 21
Non-unique Representation . . . 21
3.2.7 Term Tracing . . . 21
3.2.8 Unit Testing with ScalaTest . . . 22
3.3 Adding Indexing to *Beagle* . . . 22
3.3.1 Initial Problems . . . 22
3.3.2 Attaching an Index to ClauseSet . . . 22
3.3.3 Indexing Superposition . . . 22
3.4 Extending the Indexer . . . 22
3.4.1 Matching and Simplification in *Beagle* . . . 22
Negative Unit Simplification . . . 23
Demodulation . . . 23
3.4.2 Problems with Indexing Simplification . . . 23
3.4.3 Generalising our Fingerprint Index . . . 23
3.4.4 Applying new Indices to Simplification . . . 24
3.5 Tailoring to *Beagle*'s Hierarchic Superposition with Weak Abstraction Calculus . . . 24
3.5.1 Extending the Unification Table with Term Layers . . . 24
3.5.2 Extended Matching Table . . . 27

4 Results **29**
4.1 Beagle Before Indexing . . . 29
4.1.1 Points of Improvement . . . 29
4.2 Indexing Metrics . . . 29
4.2.1 Problem Selection . . . 29
4.2.2 Speed . . . 29
4.2.3 False Positives . . . 29
4.3 Indexing Subsumption . . . 30
4.3.1 Metric Results . . . 30

4.3.2 Comparison 30
4.4 Indexing Simplification and Matching 30
4.4.1 Further Intstrumentation 30
4.4.2 *Beagle* with Simplification Improvements 30
4.5 Tailored Improvements 30
4.5.1 Layer Checking 30
4.5.2 Metric Results 30

5 Conclusion **31**
5.1 Why this is a Very Clever Thesis 31
5.2 Future Improvements 31
5.2.1 Extended Data Structures 31
5.2.2 More Fingerprint Indices 31
5.2.3 Extensions to Fingerprint Indexing 31
Symbol count / Other Features 31
Retrieval Caching 31
Dynamic Fingerprinting 31
5.2.4 Combining Indexing Techniques 31
5.3 Final Thoughts 31

A Result Tables for TPTP Selection **33**

B Full TPTP run? **35**

Bibliography **37**

Chapter 1

Introduction

1.1 Motivation

- Describe beagle
- Advantages of beagle
- drawbacks
- some intstrumentation

1.1.1 A Theoretical Framework

Chapter 2

Background

2.1 First-Order Logic Terms and Notation

This thesis focuses around the extension of *beagle*, a *first-order logic* (FOL) theorem prover. In order to understand beagle's purpose and functions a basic understanding of the FOL logical system is required. This section provides a rudimentary overview of FOL syntax and uses; but also includes an explanation of any specialised terms and notation used throughout the paper.

2.1.1 FOL basics

Should contain

- Variables
- Symbols
- Predicates
- Quantifiers
- Notion of soundness and completeness
- Description of a 'calculus'

2.1.2 FOL with Equality and Ordering

2.1.3 Logic Terminology used in this Paper

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:

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

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

Substitution

2.1.4 Popular Problems in First Order Logic

2.1.5 The Superposition Calculus

Based on resolution, exists to allow ... [Bachmair and Ganzinger 1994]

Positive Superposition

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

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

Negative Superposition

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

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

Equality Resolution

$$\frac{s \not\approx t \vee C}{C\sigma}$$

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

Equality Factoring

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

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

These rules

2.2 Automated Reasoning and Theorem Proving

Automated Reasoning is a rapidly growing field of research where computer programs are used to solve problems stated in first order logic statements or other formal logics.

Some existing resolution theorem provers include:

2.2.1 SPASS

[Weidenbach et al. 1999]

2.2.2 Vampire

[Riazanov and Voronkov 1999]

2.2.3 E

[Schulz 2002]

2.3 Term Indexing

Term indexing is a technique used to better locate logical terms which match rules in a prover's calculus.

Top Symbol Hashing

Discriminant Trees

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 compared for unification or match compatibility. It is a *non-perfect* technique 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:

- **A**: the term has a variable at the position.

- f (any function or constant in the current system): This feature indicates that f is at the given position in the term.

- **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. We do this by simply computing which feature exists at each position. We will now revisit the example from the explanation of term positions (Section) to show the computation of a fingerprint. In the example $\{f, g\}$ are functions, $\{a, b\}$ are constants and $\{x, y\}$ are variables.

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

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

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

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

Table 2.1: Fingerprint Feature compare table for Unification [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 Matching [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

2.4.2 The Fingerprint Index

2.4.3 Position Variants

2.5 The *Beagle* Theorem Prover

The core implementation of *beagle* was developed by Peter Baumgartner et al. of NICTA. Its purpose was to demonstrate the capabilities of the *Hierarchic Superposition with Weak Abstraction Calculus*; which allows the incorporation of prior knowledge via 'background reasoning' modules. [Baumgartner and Waldmann 2013]

2.5.1 Hierarchic Reasoning

The logical calculus behind *beagle* is far from 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. Note that Waldmann continued on to co-write the paper outlining Hierarchic Superposition with Weak Abstraction [Baumgartner and Waldmann 2013].

The hierarchic reasoning system also involves an ordering on terms.

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.

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

This rule 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. They then go on to propose a new variety of abstraction where only *maximal background subterms which are neither domain elements nor variables* are abstracted. 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 clause 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 logical system. The base inference rules follow, taken directly from Section 6 of the Hierarchic Superposition with Weak Abstraction paper [Baumgartner and Waldmann 2013]. See Section 2.1.5 to compare these rules to the original superposition calculus.

Positive Superposition

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

Where (i) $\sigma =$ simple mgu (l, u), (ii) u is not a variable, (iii) $r\sigma \not\preceq l\sigma$, (iv) $t\sigma \not\preceq s\sigma$, (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

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

Where (i) $\sigma =$ simple mgu (l, u), (ii) u is not a variable, (iii) $r\sigma \not\preceq l\sigma$, (iv) $t\sigma \not\preceq s\sigma$, (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) $\sigma =$ 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) $\sigma =$ simple mgu (l, u), (ii) $r\sigma \not\preceq l\sigma$, (iii) $t\sigma \not\preceq s\sigma$, (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 mgus where abstraction variables are mapped to pure background terms.

Define and Close

2.5.4 *Beagle*'s Shortcomings

2.6 Tools Used

2.6.1 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.

2.6.2 VisualVM

2.6.3 Eclipse

Integration with Scala and ScalaTest

Chapter 3

Implementing Fingerprint Indexing

Re-iterate goals

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 and their structure which is relevant to the implementation of the Fingerprint Index.

3.1.1 Syntax and Data Structures

The first aspect of *beagle* we must examine is its existing data structures; since our term indexer must be able to understand the structure of *beagle*'s internal logical objects.

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 2.1) 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 2.1. The structure could potentially be shortened by removing the Eqn 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.

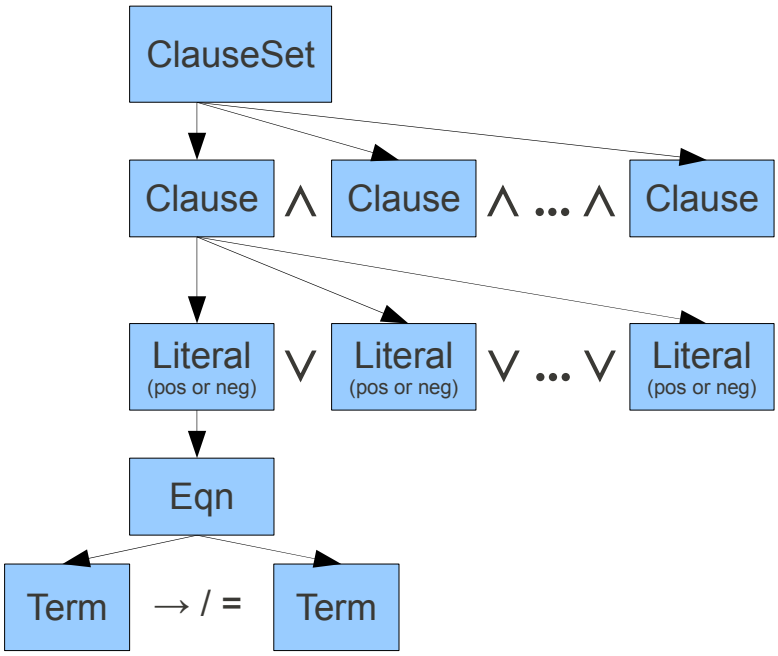

Figure 3.1: Class structure for internal representation of logical formulae.

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.

```
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. It contains all the data types and functions we will need for indexing; including 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 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 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 Fingerpint 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. A term's Fingerprint does not only match itself but also matches any compatible Fingerprints with respect to some comparison table (see Section 2.4.1). So our Index object must store Terms in a way that allows 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, and removing from the index. A logical first choice is addition; the first step of which is creating functions to sample terms at positions in order to create term fingerprints.

Listing 4 provides a function to extract a single Fingerprint Feature from a Term at the given position.

```
/** Extract the FPFeature at Position pos of the given Term object. */
def extractFeature(term: Term, pos: Position) : FPFeature = pos match {
  // Reached end of position, check symbol
  case Nil     => term match {
    case t:FunTerm => FPF(t.op) // Found function symbol, return it
    case t:Var     => FPA       // Found variable, return A
  }
  case p :: ps => term match {
    case t:FunTerm => try   {extractFeature(t.args(p), ps)}
                      //Non-existent position, return N
                      catch {case e:IndexOutOfBoundsException => FPN}
    // Found variable BEFORE end of position, return B
    case t:Var     => FPB
  }
}
```

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.

```
/** Add a Term into the given Index. Traverses Index tree
  * (adding nodes where needed) and adds Term t to a Leaf set. */
private def add (t:Term, fp:Fingerprint, index: Index):Index =
(fp, index)  match {
  //Reached a leaf at the end of the Fingerprint. Add to set.
  case (Nil,   Leaf(set)) => Leaf(t::set)
  //Still traversing tree. Add new Node or Leaf if necessary
  case (f::fs, Node(map)) => (fs, map.get(f)) match {
    //Mapping exists. Traverse through it.
    case (_,   Some(index)) => {map += (f -> add(t, fs, index))
                                Node(map)}
    //At end of Fingerprint. Create Leaf and add to it
    case (Nil, None) => {map += (f -> Leaf(List(t)))
                         Node(map)}
    //Fingerprint not over. Create Node and continue traversing
    case (_,   None) => {val newIndex:Index = new Node()
                         map += (f -> newIndex)
                         add(t, fs, newIndex)
                         Node(map)}
  }
  case (_, Node(_)) => throw new IllegalArgumentException
                ("Fingerprint is over but we are not at a leaf")
  case (_, Leaf(_)) => throw new IllegalArgumentException
                ("Reached a leaf but Fingerprint is not over")
}
```

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 Term's generated Fingerprint (relative to the Index's list of Positions); but with each recursive call of `add` we strip of one Fingerprint Feature and follow it's 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.

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 Eqn and Eqns 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 our pointer structure. The next task in building our index is allowing retrieval of

terms.

To compare two Fingerprints with each other we look at them side-by-side and check that each Fingerprint Feature shows a Y in the Fingerprint unification table from Section 2.4.1. Note that we do not need to explicitly cover each entry in the table, as it is possible to reduce it to 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.

```
/** Check two Fingerprint features for compatibility based
  * on the unification table (See page 6 of [Shulz 2012]). */
 def compareFeaturesForUnification
        (a:FPFeature, b:FPFeature) : Boolean =
 (a == b) ||
 (Set(a,b) match {
   case x if (x contains FPB) => true
   case x if (x contains FPA) => !(x contains FPN)
   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 a particular Fingerprint Index (see Section 2.4.2 and Section 3.2.1).

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

```
def retrieveCompatible
  (t: Term, fp: Fingerprint, index: Index) : TermSet =
  (fp, index) match {
//Collect all compatible (Feature,Index) pairs and continue traversal
    case (fp::fps, Node(map)) =>
        {for ((k,v) <- map if compare(fp, k))
            yield retrieveCompatible(t, fps, v)}
//Collapse all retrieved sets together with the union operator (:::)
        .foldLeft(Nil:TermSet) ((a,b) => a ::: b)
//Once we reach a leaf node
    case (Nil, Leaf(set)) => set
    case (_, Node(_)) => throw new IllegalArgumentException
         ("Fingerprint is over but we are not at a leaf")
    case (_, Leaf(_)) => throw new IllegalArgumentException
         ("Reached a leaf but Fingerprint is not over")
  }
```

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 3.4.3).

In accordance with programming best practices the final two cases in the Listing throw meaningful error messages in the case of unexpected input.

3.2.5 Matching with Subterms

In Shulz's paper the indexer he describes is only interested in the unification of full, top-level terms with other top-level terms [Schulz 2012]. Recall however the main superposition rules for *beagle*'s resolution calculus (Section 2.5.3 and [Baumgartner and Waldmann 2013]). Notice in particular the condition that s may match against a *subterm* of l. This does not match the usual required definition of unification; where we only match top-level terms.

Thus our Fingerprint Index must be able to collect all possible matches for *subterms*. To do this we will extend *beagle*'s Term object with a function collecting 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.

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

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

So our indexer will now be capable of comprehensively indexing all subterms as required. However this ability has been introduced at the cost of a couple of grave errors which, if not detected at this stage, would have destroyed the Fingerprint Index altogether.

3.2.6 Current Problems with the Fingerprint Index

At this stage we have a Fingerprint Index which is capable of comprehensively indexing a Clause all the way down to individual subterms.

Data loss

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; but this is actually no longer sufficient for our current uses. Now that we are indexing full subterms we must be able to extract the original parent term; which is impossible given only the low-level subterm representation.

Non-unique Representation

It is possible to fix our data loss issue by giving each Expression a pointer to its parent Expression (see Figure 3.1). However, as our index currently only keeps a Set of Terms, If multiple terms possessing the same subterm are ever indexed we will only keep one of them; losing any representation of the other!

3.2.7 Term Tracing

All the above issues can be solved by adding *Term Traces*.

Term tracing also solves a horrendous error introduced by matching subterms.

By indexing subterms we lose information. Must construct a trace in order to restore all required data

Note that adding subterms and their traces has resulted in a significant increase in the size of our Index object. This is negligible however since memory is cheap; and we are generally only concerned with speed.

3.2.8 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 there are issues when adding the indexer to functional use there will be no way of knowing what component is causing the problem.

3.3 Adding Indexing to *Beagle*

We now have a class fully capable

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

3.3.1 Initial Problems

Actually making use of our indexer class will require significant modification to *beagle*'s structure and proving sequence.

Superposition loop does not suit well to indexing.

Adding a single Indexer caused duplication and got messy

Refer to class and main loop flow diagrams from 3.2

3.3.2 Attaching an Index to ClauseSet

Refer to scalability, future index generalisation 3.4.3

3.3.3 Indexing Superposition

3.4 Extending the Indexer

Beagle is now fully capable of indexing its terms for superposition. There are still many ways in which this form of indexing may be improved; but it is likely to be far more effective to re-examine where *beagle*now spends most of its runtime and look into other areas which may be improved.

Refer to main section, simplification other loop.

We may confirm this suspicion by instrumenting the (now indexed) *beagle* in VisualVM. The results in Section 4.1.1 and 4.4.1 indicate (as expected) that the most significant runtime cost is now typically simplification.

3.4.1 Matching and Simplification in *Beagle*

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.

Negative Unit Simplification

$$\textbf{Negative Unit Simplification} \qquad \frac{s \not\approx t \qquad s \approx t \vee C}{C}$$

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

$$\textbf{Demodulation} \qquad \frac{l \approx r \qquad s[u] \approx t \vee D}{(s[r] \approx t \vee D)\sigma}$$

Where $\sigma =$ simple mgu (l, u).
The clause $s[u] \approx t \vee D$ may be removed.

$$\textbf{Negative Demodulation} \qquad \frac{l \approx r \qquad s[u] \not\approx t \vee D}{(s[r] \not\approx t \vee D)\sigma}$$

Where $\sigma =$ simple mgu (l, u).
The clause $s[u] \not\approx t \vee D$ may be removed.

Demodulation allows us to more quickly remove variables and terms which are *redundant*.

3.4.2 Problems with Indexing Simplification

Re-using current index produced little improvement. Cost of indexing subterms, matching against equations with $*equal*

3.4.3 Generalising our Fingerprint Index

There is a simple solution to overcome the problems listed above. Creating multiple indices. No longer restricted to the conditions of the superposition index.

This use of multiple indices obviously introduces a memory overhead. We shall argue however that this overhead is negligible for the following reasons.

Here 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.

```
/** Configuration object for a Fingerprint Index */
class IndexConfig(
  val positionsToSample : PositionList,
  val indexSubterms     : Boolean,
  val indexPureBG       : Boolean,
  val eqnToTerm         : Boolean,
  val comparator        : (FPFeature, FPFeature) => Boolean)
```

Listing 9: Class to pass settings to an arbitrary Fingerprint Index. Note that this class does not require an implementation.

- **positionsToSample**: A list of positions indicating what should be sampled to create term fingerprints.
- **indexSubterms**: Whether or not to index subterms. With this setting switched off terms are only indexed at the top level. This is very useful as subterm indexing is very slow and only required for superposition.
- **indexPureBG**: Whether or not to index pure background terms. Useful since this is not required for superposition.
- **eqnToTerm**: Whether or not to convert equations to terms. In Section 3.4.2 we pointed out that Negative Unit Simplification converts equations to terms joined by $*equal*$. Thus our Fingerprint Index must be able to index these converted terms.
- **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 Figure 3.1. Passing a different function here allows creation of separate indices for matching and unification.

3.4.4 Applying new Indices to Simplification

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

In this section we discuss the thought process in developing and implementing extensions to Fingerprint Indexing in order to better tailor it to *beagle*'s rather unique logical calculus.

3.5.1 Extending the Unification Table with Term Layers

In the Hierarchic Superposition with Weak Abstraction Calculus all terms have a concept of being 'Foreground' or 'Background'. In Section 2.5 we discussed this concept; referring to it as the *layer* of a term. It is worth noting at this stage that

computing the layer of a term 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:

- 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 *background* or *abstraction* sorted variables. These variables can only be used for pure background terms; a fact we may use to restrict the possible matches for unification.

The layered-ness of function symbols is also relevant to our comparison. $f+$ in the following table signifies a position where the entire subterm from this position downwards is 'pure background'. Keep in mind that this definition is slightly different to the definition for **A**+ and **B**+; as we must consider all function symbols below f itself.

At this point it is important to note that these added fingerprint features slightly modify the original **A**, **B** and f features. These features will now only represent the foreground layered positions.

Table 3.1 displays the unification table with our new background feature symbols. The table has grown 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 matches for unification; extended by considering term layers.

	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.

We will justify the new section of the table line by line:

- 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*' 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**.

To go with this table we present its corresponding Scala matching code in Listing 10. Unfortunately the steep increase in table size results in the amount of code required exploding. It also becomes impossible to use our earlier trick of Set matching (from Listing 6); due to the need for parameterised Fingerprint symbols (i.e. **A**+ and **B**+ represented as FPA(true) and FPB(true)).

```
/** Check two Fingerprint features for compatibility based
  * on the *extended* unification table (See table in report).*/
 def compareFeaturesForUnification
     (a:FPFeature, b:FPFeature) : Boolean =
 (a,b) match {
   case (FPF(f1), FPF(f2))     => (f1.op == f2.op) &&
                                  (if (f1.isFG || f2.isFG)
                                      (!f1.isPureBG && !f2.isPureBG)
                                   else true)
   case (FPF(f), FPB(true)) => f.isPureBG
   case (FPB(true), FPF(f)) => f.isPureBG
   case (_, FPB(_))         => true
   case (FPB(_), _)         => true
   case (FPF(f), FPA(true)) => f.isPureBG
   case (FPA(true), FPF(f)) => f.isPureBG
   case (FPN, FPA(_))       => false
   case (FPA(_), FPN)       => false
   case (_, FPA(_))         => true
   case (FPA(_), _)         => true
   case (FPN, FPN)          => true
   case _                   => false
 }
```

Listing 10: Scala code to extract fingerprint features for extended layer matching.

3.5.2 Extended Matching Table

Chapter 4

Results

4.1 Beagle Before Indexing

We have stated several times throughout this report that the key area of improvement for *beagle*is in clause resolution. However we have yet to provide any evidence to that fact. Here we provide some results from initial investigations (before Fingerprint Indexing was implemented) used to identify the key areas of improvement for *beagle*.

4.1.1 Points of Improvement

Refer to results for instrumentation; showing what may be improved with indexing.

4.2 Indexing Metrics

4.2.1 Problem Selection

Not going to run everything against all of TPTP. Will take out a subset of TPTP (25–50 problems) run them against this set. Show the set and justify inclusions/exclusions

4.2.2 Speed

4.2.3 False Positives

Explain this metric and how it impacts performance.

Not as good results due to extreme other conditions on inference rules. refer to differences in background rules. Also matching subterms! Cheap throwaway FPs are not a concern. totally useless even. Comment on change in how they are measured.

It is possible to naiivley boost FP to 0 with perfect indexing; but this would not yield any speed improvement. Balancing FPs with fingerprint length is key.

4.3 Indexing Subsumption

4.3.1 Metric Results

We observe many, even after a myriad of optimisations. Notice that this is due to the structure of beagle; many retrieved terms are cheaply thrown out due to other conditions (such as being a parent term, being ordered, etc.) and do not significantly impact performance.

Refer to email from Stephan. compare false positive results

4.3.2 Comparison

4.4 Indexing Simplification and Matching

4.4.1 Further Intstrumentation

4.4.2 *Beagle* with Simplification Improvements

4.5 Tailored Improvements

4.5.1 Layer Checking

4.5.2 Metric Results

Chapter 5

Conclusion

5.1 Why this is a Very Clever Thesis

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 cut due to time constraints.

5.2.1 Extended Data Structures

The focus of this thesis has been with high-level logical improvements. Add some low-level speed 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 barley ever) but creates memory EXPLOSION

5.2.2 More Fingerprint Indices

5.2.3 Extensions to Fingerprint Indexing

Symbol count / Other Features

Retrieval Caching

Dynamic Fingerprinting

5.2.4 Combining Indexing Techniques

5.3 Final Thoughts

Appendix A

Result Tables for TPTP Selection

Appendix B

Full TPTP run?

Bibliography

Bachmair, L. and Ganzinger, H. 1994. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput. 4*, 3, 217–247. (p. 4)

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. (p. 7)

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. 7, 8, 20)

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. (p. 5)

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. 3, 5, 6, 15, 16, 17, 19, 20, 25)

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)