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

Typeset in Palatino by TEX and LATEX 2ε.

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

Timothy Clarence Richard Cosgrove
3 October 2013

For Dana.

Acknowledgements

Thank you to my Supervisor and all...

Abstract

This should be the abstract to your thesis...

Contents

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 Important Terminology

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

Subsumption

Unification

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] [Asperti and Tassi 2010]

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 clauses and terms for which inference rules in a calculus will apply. In particular, indexers are used to find all terms which will or are likely to *unify* or *match*; which is a typical condition of any inference rule (for example all rules for the superposition calculus in Section 2.1.5 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 comparison and examples we present here some other techniques.

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.

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 store terms in a

Path Indexing

Path Indexing is actively used by the Vampire theorem prover [Riazanov and Voronkov 1999]

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. Like 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:

- **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.

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

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

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

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

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. This process subtly indicates the key difference between the Fingerprint Index and a Discrimination Tree: since all term fingerprints are the same length, the Index has a fixed depth.

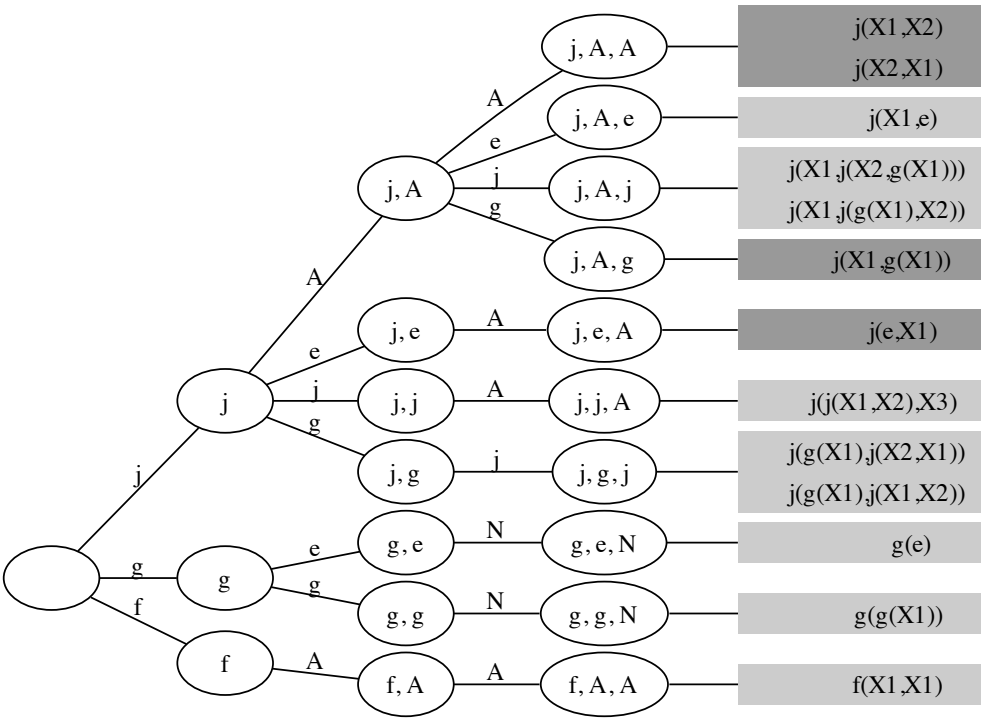

Figure 2.1: Example structure of a Fingerprint Index. Taken unmodified from [Schulz 2012, p7].

2.4.3 Comparing Fingerprints and Retrieving

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**

2.4.4 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.

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

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. 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) $r\sigma \not\succeq l\sigma$, (iv) $t\sigma \not\succeq s\sigma$, (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) $r\sigma \not\succeq l\sigma$, (iv) $t\sigma \not\succeq s\sigma$, (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) $r\sigma \not\succeq l\sigma$, (iii) $t\sigma \not\succeq s\sigma$, (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.

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.

[École Polytechnique Fédérale de Lausanne (EPFL) 2013]

2.6.2 VisualVM

[Oracle Corporation 2013]

2.6.3 Eclipse

Integration with Scala and ScalaTest

[Eclipse Foundation 2013] [Dragos 2013] [Venners et al. 2013]

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.

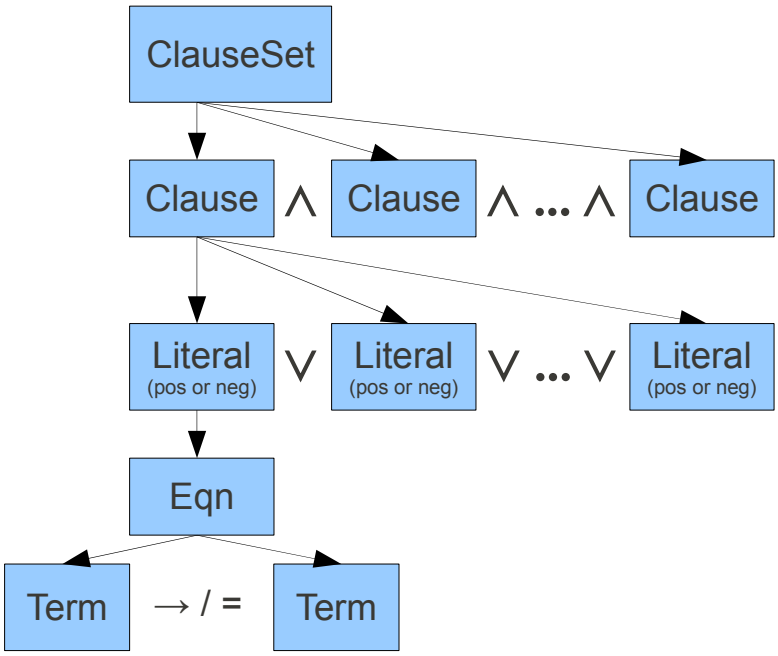

Figure 3.1: Class structure for internal representation of logical formulae.

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. 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 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 its pointer structure. The next task in building our index is allowing retrieval of 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.

```
/** 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 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.

```
def retrieveCompatible (fp: Fingerprint, index: Index) : TermSet =
(fp, index) match {
//Collect all compatible (Feature,Index) pairs and continue traversal
    case (f::fs, Node(map)) =>
        {for ((k,v) <- map if compare(f, k))
            yield retrieveCompatible(fs, v)}
//Collapse all retrieved sets together with the union operator (:::)
        .foldLeft(Nil:TermSet) ((a,b) => a ::: b)
//Once we reach a Leaf we simply return the compatible set
    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 3.4.2).

`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 Shulz'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 s may match against a *subterm* of l (this condition also exists in the original superposition calculus, Section 2.1.5). 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.

```
/** 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.

With this extension in place we can index subterms by extending our addition function (Listing 5) to also add all subterms. So our indexer will now be 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 sec:unittest) 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 are 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. Ensuring that these pointers were correct at all times however 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.

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 Eqn, 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 (after adding the Index to *beagle*) there are issues we will have no way of knowing what component is causing the problem.

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 solution exists.

Rather than attach a single Fingerprint Index to the entire *beagle* inference process, we instead add an Index to the ClauseSet class. Clauses from `new` are removed one by one and possible made redundant before moving to `old`. `old` itself however is more static; it only ever has clauses *added* to it. By only indexing the `old` set we ensure no redundant clauses appear in inferences; and it becomes easy to only use the `old` index 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; but there is still significant modification required to use Index 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
$$\frac{l \approx r \vee C \qquad s[u] \approx t \vee D}{\text{abstr}((s[r] \approx t \vee C \vee D)\sigma)}$$

Where (i) $\sigma =$ simple mgu (l, u), (ii) u is not a variable, (iii) $r\sigma \not\succeq l\sigma$, (iv) $t\sigma \not\succeq s\sigma$, (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) $r\sigma \not\succeq l\sigma$, (iv) $t\sigma \not\succeq s\sigma$, (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 . 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 ls 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 try either side as l.

Once we have found each usable l we simply retrieve all terms compatible for unification (see Section 3.2.4) and use *beagle*'s existing superposition code to confirm unification and check the other inference rules conditions.

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 with a subterm u. As in the from case we must find all possible terms which we can use for s; 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.

Once we have a subterm for u we retrieve compatible Terms from the index. Unlike the from case we are not done here, l may only be a top level term so we must actually discard any subterms. 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 4.1.1 and 4.4.1 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 Simplfication* 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:

Negative Unit Simplification

$$\frac{l \not\approx r \qquad s \approx t \vee C}{C}$$

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 *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 $l \to r$ refers to an ordered Literal $l \approx r$ and $s[u]$ refers to a Term s with a subterm u).

Demodulation

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

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.

Negative Demodulation

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

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 if we have some unit Clause $l \to r$. 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 $\$equal$:

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

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 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 implemenation. 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. As we have mentioned before however 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; but they are unlikely to cause more than a few unnecessary inferences and a minor 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'. 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].

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

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

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 11. 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 11: Scala code to extract fingerprint features for extended layer matching.

3.5.2 Extended Matching Table

3.5.3 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 encounter. 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`.

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

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

5.2.4 Combining Indexing Techniques

5.3 Final Thoughts

Appendix A

Result Tables for TPTP Selection

A.1 Unmodified *Beagle*

A.2 Including Superposition Indexing

A.3 Including Simplification Indexing

A.4 Including Background/Foreground Checks

Appendix B

Results when Varying Fingerprint Sample Positions

B.1 Sample 1

B.2 Sample 2

B.3 Sample 3

B.4 Sample 4

B.5 Sample 5

B.6 Sample 6

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. 4)

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. 9)

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. 9, 10, 22, 25)

Dragos, I. 2013. Scala IDE for Eclipse. `http://scala-ide.org/`. (p. 11)

Eclipse Foundation. 2013. Eclipse homepage. `http://www.eclipse.org/`. (p. 11)

École Polytechnique Fédérale de Lausanne (EPFL). 2013. Scala homepage. `http://www.scala-lang.org/`. (p. 11)

Oracle Corporation. 2013. VisualVM All-in-One Troubleshooting Tool. `http://visualvm.java.net/`. (p. 11)

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, 6)

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, 6, 7, 8, 17, 18, 19, 21, 22, 32)

Venners, B., Berger, G., and Seng, C. 2013. Scalatest homepage. `http://www.scalatest.org/`. (p. 11)

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. 28)