Software retrieval based on semantic properties.

Khaled. Khalil

University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
https://scholar.uwindsor.ca/etd/3510
NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.
Software Retrieval Based on Semantic Properties

by
Khaled Khalil

A Thesis
Submitted to the Faculty of Graduate Studies and Research through the School of Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Science at the University of Windsor

Windsor, Ontario, Canada
1995
The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

Abstract

Software reuse is the process of using existing software components during the construction of software systems in order to reduce the effort and maintenance of the system. The growth of reuse of software repositories has led to the design of mechanisms to retrieve reusable components. This thesis presents the design and implementation of an automated software retrieval where components can be automatically stored and retrieved using informal partial specification such as semantic properties of functions.
To my father
my mother
my sisters
my brothers
Acknowledgements

First, I would like to thank God for giving me the strength and the ability to complete this work.

I would also like to acknowledge Dr. Y. Park for his supervision, advice, and patience during the progress of this thesis, Dr. R. Frost for his comments, and for the facilities in the department, and Dr. H. Kwan for serving as a member of the committee.

Special thanks to my parents for their encouragement, my sisters for their patience, and my brothers for their sacrifices.

Finally, warm thanks to my graduate colleagues and friends for their support and tips which helped in the completion of this work.
TABLE OF CONTENTS

Abstract ......................................................... iv
Acknowledgements .............................................. vi
List of Figures .................................................. x
List of Tables ................................................... xii

1 INTRODUCTION ................................................. 1
  1.1 Background ................................................. 1
  1.1.1 Syntactic Approach .................................. 3
  1.1.2 Semantic Approach .................................. 3
  1.1.2.1 Type Signature ................................ 4
  1.1.2.2 Sampling Behavior .............................. 4
  1.1.2.3 Formal Specification ......................... 5
  1.2 Motivation and Goal .................................. 5
  1.3 Our Approach ........................................... 7
  1.3.1 Thesis Statement ................................ 9
  1.3.2 Importance of the Thesis ....................... 10
  1.4 Organization of the Thesis ....................... 10

2 Semantic Properties of Components ..................... 12
  2.1 Strictness Information ............................... 12
  2.2 Lifetime Information ................................. 20
2.3 Length Comparison Information .......... 33
2.4 One-to-one Information .................. 38
2.5 Onto Information ........................ 41

3 Retrieval Based on Semantic Properties ........ 45
  3.1 Retrieval with One Argument ............. 45
  3.2 Retrieval with Two Arguments .......... 47
  3.3 Discussion ................................ 51

4 WISER+: A Prototype System ................. 56
  4.1 Overview of WISER ....................... 56
  4.2 WISER+: A New Approach ............... 57
     4.2.1 Insertion ............................ 58
     4.2.1.1 EXAMPLE: .......................... 60
     4.2.2 Deletion ............................. 62
     4.2.2.1 EXAMPLE: .......................... 63
  4.2.3 Retrieval ............................. 64
     4.2.3.1 EXAMPLE: .......................... 65
  4.2.4 Browser ............................... 72
     4.2.4.1 EXAMPLE ........................... 73
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Retrieval Approaches</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>User Defined Function</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Semantic Retrieval</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>Non-list Strictness Information</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>List Strictness Information</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>Notation of List Strictness Information</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>Lifetime Information</td>
<td>23</td>
</tr>
<tr>
<td>8</td>
<td>Notation of Non-list Lifetime Information</td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td>List Type Lifetime Information</td>
<td>27</td>
</tr>
<tr>
<td>10</td>
<td>Notation of List Type Lifetime Information</td>
<td>28</td>
</tr>
<tr>
<td>11</td>
<td>Length Comparison Information</td>
<td>34</td>
</tr>
<tr>
<td>12</td>
<td>One-to-one Information</td>
<td>39</td>
</tr>
<tr>
<td>13</td>
<td>One-to-one Information of Sort Function</td>
<td>41</td>
</tr>
<tr>
<td>14</td>
<td>Onto Information</td>
<td>42</td>
</tr>
<tr>
<td>15</td>
<td>Intersection Function</td>
<td>48</td>
</tr>
<tr>
<td>16</td>
<td>Overview of WISER</td>
<td>57</td>
</tr>
<tr>
<td>17</td>
<td>WISER+ Main Menu</td>
<td>58</td>
</tr>
<tr>
<td>18</td>
<td>Insertion Process</td>
<td>59</td>
</tr>
<tr>
<td>19</td>
<td>Insertion Dialog (Type &amp; Semantic Properties)</td>
<td>61</td>
</tr>
<tr>
<td>20</td>
<td>Error Message of Insertion</td>
<td>61</td>
</tr>
<tr>
<td>21</td>
<td>Deletion Process</td>
<td>62</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>22</td>
<td>Deletion Dialog</td>
<td>63</td>
</tr>
<tr>
<td>23</td>
<td>Retrieval Process</td>
<td>64</td>
</tr>
<tr>
<td>24</td>
<td>Retrieve Dialog (Type Only)</td>
<td>67</td>
</tr>
<tr>
<td>25</td>
<td>Exact Match of Type Search Key</td>
<td>68</td>
</tr>
<tr>
<td>26</td>
<td>Semantic Properties Retrieval Dialog</td>
<td>69</td>
</tr>
<tr>
<td>27</td>
<td>Exact Match (Type &amp; Semantic Properties)</td>
<td>69</td>
</tr>
<tr>
<td>28</td>
<td>General Match (Type &amp; Semantic Properties)</td>
<td>70</td>
</tr>
<tr>
<td>29</td>
<td>Freezing Match (Type &amp; Semantic Properties)</td>
<td>71</td>
</tr>
<tr>
<td>30</td>
<td>Specific Match (Type &amp; Semantic Properties)</td>
<td>71</td>
</tr>
<tr>
<td>31</td>
<td>Browser Process</td>
<td>72</td>
</tr>
<tr>
<td>32</td>
<td>Browser Window</td>
<td>73</td>
</tr>
<tr>
<td>33</td>
<td>Semantic Properties of Difflist (Shown in the Lower Section in Browser Window)</td>
<td>74</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Strictness Information of some Miranda Functions</td>
<td>16</td>
</tr>
<tr>
<td>Table 2</td>
<td>Strictness Information for Length Function</td>
<td>19</td>
</tr>
<tr>
<td>Table 3</td>
<td>Strictness Information of Sumlist Function</td>
<td>20</td>
</tr>
<tr>
<td>Table 4</td>
<td>Strictness Information of Some Miranda Functions</td>
<td>20</td>
</tr>
<tr>
<td>Table 5</td>
<td>Lifetime Information of Insert Function</td>
<td>25</td>
</tr>
<tr>
<td>Table 6</td>
<td>Lifetime Information of Min Function</td>
<td>26</td>
</tr>
<tr>
<td>Table 7</td>
<td>Lifetime Information of Add Function</td>
<td>27</td>
</tr>
<tr>
<td>Table 8</td>
<td>Lifetime Information of TI, Reverse, and Rduplicate Functions</td>
<td>30</td>
</tr>
<tr>
<td>Table 9</td>
<td>Lifetime Information of Largest, Inter, and F Functions</td>
<td>31</td>
</tr>
<tr>
<td>Table 10</td>
<td>Lifetime Information of Difflist Function</td>
<td>33</td>
</tr>
<tr>
<td>Table 11</td>
<td>Length Comparison Information of Sort, Reverse, and TI Functions</td>
<td>35</td>
</tr>
<tr>
<td>Table 12</td>
<td>Length Comparison Information of TI, Sort and Rduplicate Functions</td>
<td>36</td>
</tr>
<tr>
<td>Table 13</td>
<td>Length Comparison Information of Concat &amp; CondCon Functions</td>
<td>37</td>
</tr>
<tr>
<td>Table 14</td>
<td>One-to-one information of ABS and fact functions</td>
<td>40</td>
</tr>
<tr>
<td>Table 15</td>
<td>One-to-one Information of Reverse, Sort, and TI Functions</td>
<td>40</td>
</tr>
<tr>
<td>Table 16</td>
<td>Onto Information of ABS, and Fact Functions</td>
<td>43</td>
</tr>
<tr>
<td>Table 17</td>
<td>Semantic Properties of the Reverse Miranda Function</td>
<td>46</td>
</tr>
<tr>
<td>Table 18</td>
<td>Semantic Properties of Some Similar Type Function</td>
<td>47</td>
</tr>
<tr>
<td>Table 19</td>
<td>Semantic Properties of the Intersection</td>
<td>49</td>
</tr>
<tr>
<td>Table 20</td>
<td>Semantic Properties of Some Miranda Functions</td>
<td>50</td>
</tr>
<tr>
<td>Table 21</td>
<td>Primelist Miranda Program</td>
<td>75</td>
</tr>
<tr>
<td>Table 22</td>
<td>Magic Square Matrix</td>
<td>79</td>
</tr>
<tr>
<td>Table 23</td>
<td>Magic square Miranda Program</td>
<td>80</td>
</tr>
</tbody>
</table>
Chapter 1  INTRODUCTION

1.1 Background

Since the birth of computer programming, many software developments have been originally created written line by line. One could find a more beneficial way to build new software by using pre-existing components previously created software can be applied for different purposes. This approach is known as software reusability, which is defined by Ausint [12] as:

"The capability of software components to be used again or repeatedly in applications other than the one for which it was originally developed. In order to be effectively reused, the components may have to be adapted to the requirements of the new application."

The traditional method of writing new software developments consumes a great deal of time and is very costly. In order to have an efficient approach, software developers are favoring the idea of software reuse because of the reduced time consumption and decreased effort in creating software components. According to Huff, Thomson, Gish [3] "software reuse has the potential to increase productivity and reduce time to make through savings in design, code and test effort, as well as to improve quantity and reduce risk through the use of proven components."
In order to show how software reuse is successful and important, Prieto-Diaz [12] reported that the missile system division of Raythen observed a six-year reuse program. He found that 60% of all business and application designs and codes could be reused, giving a net of 50% increase in productivity. Fujitsu's software development for an electronic switching system took a simpler approach, called the Fujitsu Information Support Center (ISC). Fujitsu reported that 70% of their 300 projects now run to schedule compared with 20% before the ISC. Finally, the GTE data system approach reported a similar increase in productivity through reuse. Hence the importance of the software is illustrated through the following benefits:

1. *Saves time.*

2. *Saves money.*

3. *Requires less effort.*

4. *Has reliable components.*

The process of reusing components can be divided into three steps: **Storage, Retrieval, and Adaptation/Integration.** The first step (Storage) is the process of organizing and storing the components, each with its own abstract information that later will be helpful in identifying the user desired component. The next step is the retrieval process that will be used to find the exact or similar components to the needed one with respect to supplied user's abstract information. The final step is the adaptation/integration process. The components that have been found in the retrieval process
Software Retrieval Based on Semantic Properties

will be inspected and modified if needed, to be integrated into the new implementation.

The work of this thesis is only concerned with the retrieval process. This process can be specified by two approaches: **Syntactic and Semantic**.

1.1.1 Syntactic Approach

The syntactic approach method bases its search key on the name of the component. The user has to define a close function name that he/she wants to implement, then search through the repository library to find some candidate functions that have the same name or that are slightly different.

The syntactic approach is an easy method to implement, and also it is simple for the user to define his search key. But on the other hand, the candidate for retrieval components which are similar to the user’s defined function might not be appropriate components that is because the retrieval search has no other information than the name of the function.

1.1.2 Semantic Approach

The semantic approach, which deals with the meaning of the components, is the most promising approach since it allows the user to retrieve more accurate components than the syntactic approach. This approach relies on more information than the function name. This is why most of the
researchers focus on the semantic approach in their investigation of an automatic retrieval of reusable software.

The semantic approach can be divided into three techniques of retrieval as shown in Figure 1.

**Figure 1 Retrieval Approaches**

![Retrieval Approach Diagram]

---

**Type Signature**

Type Signature consists only of defining the type of the function to be used as a search key. The user of such a system is only required to know the type information of such a function in order to retrieve any exact or similar candidate components. This retrieval results in searching the library for functions basically matching the type of the user desired components.

**Sampling Behavior**

The Sampling Behavior approach is concerned only with the behavior of the executable components. This retrieval technique actually executes the components in the library with the sample input given by the user. This approach retrieves only all the candidate components that have an exact
or similar behavior (input-output relationship) to the function that the user needs.

**Formal Specification**

Formal Specification is considered to be the ideal approach to retrieve more precise and accurate reusable components. The Specification approach is based on mathematical notations that are used to define the function properties. In general, Formal Specification is undecidable and unimplementable due to the infinite and different number of properties that functions can have, and the impossibility of matching any two functions using these infinite different properties.

1.2 Motivation and Goal

The growth of software developments has led the repository library to become very immense and it contains much reusable software. The user naturally prefers to have an easy way to retrieve more accurate components that suit his/her desired function with less time and effort. The available approaches do not make an easy task of retrieving accurate components. Each has its own problems which will affect the retrieval process of the most promising candidate components.

Type signature is actually better than syntactic approach for using the type information instead of the function name as a search key. On the other hand,
this approach comes with the cost of retrieving lots of functions that have a similar type. It is then the user's job to inspect each one of them to find out the components that suit his/her implementation need. In addition, type approach does not give accurate components because the search only depends on the type and no other information that is related to the components. Hence the problem of the signature approach can be summarized as follow:

1. *Retrieves many components that have to be inspected by the user.*

2. *Does not give accurate components.*

Full formal specification, as we mentioned earlier, is an ideal approach to retrieve the most accurate and promising candidate components. As is commonly known, formal specification is based on the mathematical notations that are used to define the function properties. Therefore a real expert is needed to write the formal specification of any function. In addition, each function can be defined with infinite and different numbers of properties. These can make it impossible to match any two identical components, and can also lead to the undecidable problem. Thus, the problems of full formal specification retrieval approach can be summarized as follow:

1. *Matching full formal specification of components is often impossible.*

2. *Impossible to implement with full formal specification.*
3. An expert is always needed to specify a full formal specification of the components.

However, problems with the above approaches can lead us to determine a solution that can be beneficial and result in a proper retrieval approach. To find a better and more useful approach than the type signature, but less difficult than the full formal specification is our goal: to retrieve more useful components with less formal specification.

1.3 Our Approach

The objective of this thesis is to define a new retrieval approach that can be useful to achieve more accurate and useful components. Our approach is to define the number of semantic properties of functions. This latter can be used to locate and retrieve more useful and accurate components than a based type and is at the same time less formal than full formal specification. This thesis focuses on defining the semantic properties which are Strictness, Lifetime, Length Comparison, One to One, and Onto (which will be discussed later), to determine how these properties are helpful in our retrieval approach.
Our Semantic Retrieval Approach works as follow: first of all, each one of the components in the repository library must be defined with the properties already mentioned. The user then defines the semantic properties for a function that he/she wants to implement as shown in Figure 2. Next, the user must query the library for components that match his/her defined properties. The retrieval process first gets all the components that match the type of the function queried. Then the Semantic Properties Matching Technique will be applied to these functions, which results in retrieving only the components that have similar semantic properties as shown in Figure 3.
1.3.1 Thesis Statement

The work of this thesis is concerned with:

1. *Investigation of effective methods of retrieving reusable software components using informal partial specification (semantic properties) of components.*

2. *Based on the method, implementation of a prototype system for retrieving components in a functional programming environment like Miranda.*

3. *Perform some testing components to show how this method is effective and results in retrieving more accurate functions that satisfy user’s requirements.* (Eliminate unneeded components.)
1.3.2 Importance of the Thesis

The importance of the thesis is that it introduces a New Retrieval Approach that achieves a more accurate set of component with less formal specifications. The reason that this new approach is important can be explained by the following:

1. *Uses less formal specification.*

2. *Increases precision and retrieves more accurate and useful components than type search key.*

3. *Easier to use than full formal specification approach.*

4. *No need for expertise on full formal specifications.*

1.4 Organization of the Thesis

This thesis is organized as six chapters. Chapter 1 states a general overview of software reusability and the approach to retrieve more accurate components.

Chapter 2 deals with definitions of the semantic properties and explains how each functions.

Chapter 3 demonstrates the examples of the Semantic Properties Retrieval Approach.

Chapter 4 states an overview of WISER+ and explains through examples how retrieval, insertion, deletion, and browser works.
Chapter 5 provides two applications that illustrate how using Semantic Properties Retrieval can help in eliminating the unneeded retrieved components.

Finally, chapter 6 states the conclusion and future work.
Chapter 2  Semantic Properties of Components

Semantic properties are more helpful for defining and retrieving accurate components than Based Type Retrieval, from the repository library. The new retrieval approach uses simple defined properties like Strictness, Lifetime, Length Comparison, One to One, and Onto. Each component in the repository library has to be defined by using these properties which will aid in the retrieval process in order to find the most promising candidate components that the user desires.

To define the library components with these semantic properties, one should know how each of these properties work. This chapter focuses in detail on each of these semantic properties using an example to make it comprehensible. Hopefully by the end of this chapter the reader will be familiarized and be more comfortable with the given definition of the above properties.

2.1 Strictness Information

Strictness is one of the most important properties that could be used to define a specific behavior of functions. The following function:

\[ F \ X \ Y = \begin{cases} X, & X > 0 \\ Y, & Otherwise \end{cases} \]
is strict in the argument X but not in Y. Hopefully by the end of this section the reader will be able to identify why the strictness of a such function is so.

**WHAT?**

Strictness is defined as: "A function is strict in its n arguments if a call to the function can be evaluated only if all n arguments can be evaluated"[1]. In other words, Strictness information defines whether a function needs the value of its arguments in order to be evaluated.

**WHY?**

Strictness is one of the function properties that could be used to define the behavior of the function. Applying strictness analysis on the function results in an important property that is used in software retrieval. This property can help to differentiate between different functions in the repository library.

For example, suppose one wants to retrieve a function that adds its two arguments. In the library, there are two function F1 and F2 with the same type of:
\begin{align*}
\text{num} & \rightarrow \text{num} \rightarrow \text{num}\\
\text{where:} & \\
F1 \ X \ Y & = X + Y\\
\text{and} & \\
F2 \ X \ Y & = X, \ X > 0\\
& = Y, \text{Otherwise}
\end{align*}

F1 is strict in both arguments X and Y. On the other hand, F2 is strict in X but not in Y. We know that the function we want to retrieve has to be strict in both arguments; thus using strictness information, we only need to retrieve function F1 which has the same property (strictness information) as the function desired.

Therefore, we can conclude that using a strictness property could be helpful in distinguishing between functions and can also be used in software retrieval to help identify more accurate components.

The analysis of the strictness information depends on the argument types which are divided into 2 cases: Non-list and List type.
Non-list type:

Figure 4  Non-list Strictness Information

Strictness Information

- Needed-Value
- Non-needed Value

All arguments that are not of List type can be defined with strictness information by analyzing the arguments of the function to determine the argument value which is needed. Figure 4 shows how strictness information of non-list arguments can be illustrated. Either the value is needed or it is not. The following examples show how Strictness information can be found.

Consider the following function:

\[
\text{\textit{min}} \ X \ Y = \begin{cases} 
X, & X \leq Y \\
Y, & \text{Otherwise} 
\end{cases}
\]

This function finds the minimum of two arguments. In order to do so, both arguments have to be compared. The function \textit{min} needs to know the value of \(X\) and \(Y\) before giving any result. Hence, \textit{min} is strict in both arguments \(X\) and \(Y\).
Now consider another function:

\[ F \ X \ Y = X, \ X > 0 \]

\[ = Y, \text{Otherwise} \]

Function F outputs the first argument X if X is positive. Otherwise, it returns Y. In this case, F only checks for the value of X. Therefore, F is strict in X but not in Y.

Table 1 contains the Strictness Informations of the above given functions.

Where:

Y:: means that the function needs the argument value.

N:: means that the function does not need the argument value.

<table>
<thead>
<tr>
<th>num -&gt; num -&gt; num</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strictness Information</strong></td>
</tr>
<tr>
<td><strong>FUNCTION-NAME</strong></td>
</tr>
<tr>
<td>MIN</td>
</tr>
<tr>
<td>F</td>
</tr>
</tbody>
</table>

**List Argument:**

The List Argument case is more complicated than Non-list. In this case, not only do we have to find the strictness information of the List, but also
the components value of the list and how much of these values are needed.

Figure 5 List Strictness Information

![Strictness Information Diagram]

Figure 5 shows the process of constructing the strictness information of a List argument. First, we check if the list is needed. In case it is not, we stop there; otherwise, we have to proceed further in order to find out how much of the components values are needed, whether it's all, only some, or none of them. Before presenting any examples, some notations that will be used later (shown in Figure 6) are defined below:
Figure 6 Notation of List Strictness Information

Strictness Information

Y

N

A

S

Z

Y:: List is needed.

N:: List is not needed.

A:: All the components values in the list are needed.

S:: Only some of the component values in the list are needed.

Z:: None of the component values in the list is needed.

Now consider the following example:

\[ \text{Length } [ ] = 0 \]

\[ \text{Length } (a : as) = 1 + \text{Length } as \]

This function takes one argument of the list type and returns the length of that list. In this case, the list is needed in order to evaluate the function.
On the other hand, the computation of the length of the list does not need the actual value of the components of the list. Therefore, the strictness information is summarized in Table 2 below.

Table 2 Strictness Information for Length Function

<table>
<thead>
<tr>
<th>Strictness Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a] -&gt; num</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FUNCTION-NAME</th>
<th>ARG</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>Y</td>
</tr>
</tbody>
</table>

Another example:

```
sumlist [] = 0

sumlist (a : as) = a + sumlist as
```

This function takes a list of type num and returns the summation of the elements of that list. In order to evaluate the summation of this particular list, we must have the list and also need to know the actual value of all components in that list. Thus, the strictness information of the function sumlist is that the list is needed and the components value as well (as shown in table 3).
Table 3 Strictness Information of Sumlist Function

<table>
<thead>
<tr>
<th>[num] -&gt; num</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strictness Information</strong></td>
</tr>
<tr>
<td><strong>FUNCTION-NAME</strong></td>
</tr>
<tr>
<td>sumlist</td>
</tr>
</tbody>
</table>

Using the same analysis and the same notation, we built Table 4 for some given functions and their strictness information.

Table 4 Strictness Information of Some Miranda Functions

<table>
<thead>
<tr>
<th>[a] -&gt; [a]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Strictness Information</strong></td>
</tr>
<tr>
<td><strong>FUNCTION-NAME</strong></td>
</tr>
<tr>
<td>sort</td>
</tr>
<tr>
<td>reverse</td>
</tr>
</tbody>
</table>

2.2 Lifetime Information

Another important property, Lifetime can be useful in the process of Software Retrieval. Lifetime information defines whether arguments of a particular function can be returned after a function call. Given the following function:

\[ tl \ (a : as) = as \]
we could say that its argument outlives the function call.

WHAT?

Lifetime is defined as: "The lifetime information is about whether any arguments of a function can partially or completely outlive its function call." This means that any arguments of a function might be included partially or completely in the output after a function call.

WHY?

Functions can behave differently. Lifetime is one of the semantic properties that can be used to define the behavior of a function. Using this information can help to distinguish between functions that results in retrieving more accurate components. For example, suppose we want to retrieve a function that takes two arguments of type `num` and returns the addition of these two arguments. In the library established, there are two functions of similar type:

\[ num \rightarrow num \rightarrow num \]

where:

\[ Add \ X \ Y = X + Y \]

and

\[ F \ X \ Y = X , X > 0 \]

\[ = Y , Otherwise \]
where

X or Y in Add function does not appear in the output. And where,

X or Y in F function does appear in the output.

We know that the function we want to retrieve must not have any of its two arguments appear in the output. Based on this lifetime property information, we only have to retrieve the matching Add function. Hence, the user can benefit from using lifetime information to differentiate between different functions and retrieve more useful components from the repository library.

Define Lifetime Information:

Similar to strictness information, Lifetime divides its argument types into two cases: Non-list and List type.

Non-list Type:

Non-list type argument can be defined with lifetime information by analyzing the function call in its arguments. This is so in order to find out whether any of the arguments can possibly outlive the function call.
Figure 7 shows how the lifetime property can be described. The argument of a function can either Always, Sometimes, or Not-at-all be included in the output after a function call.

The following examples describe how each of these properties can be found. Some notations that will be used later in defining the lifetime property...
are defined below and shown in Figure 8. (Figure 7 is the same as Figure 8 but with different notations.)

Where:

A:: Argument is always included in the output.

S:: Argument is sometimes included in the output, depends on the function itself.

N:: Argument is not always included in the output.

Non-list type property is divided into 3 cases: Always, Sometimes, and Not-at-all.

**Always Case:**

Always case of a Non-list property means that the argument is always included in the output after a function call.

Consider the following example:

\[ \text{insert } X \ [\ ] = [X] \]

\[ \text{insert } X \ (Y : YS) = X : Y : YS, \quad X \leq Y \]

\[ = Y : \text{insert } X Y S, \quad \text{Otherwise} \]
In this example, there is only going to be taken the Non-list argument and the List argument will be taken later. *Insert* function takes two arguments \(X\) and a list and returns a list that includes argument \(X\). Thus, \(X\) is always included in the function output. Lifetime property information is shown in Table 5.

<table>
<thead>
<tr>
<th>a (\rightarrow) [a] (\rightarrow) [a]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lifetime Information</strong></td>
</tr>
<tr>
<td><strong>FUNCTION-NAME</strong></td>
</tr>
<tr>
<td>insert</td>
</tr>
</tbody>
</table>

**Sometimes Case:**

The Sometimes Case property means that the argument can sometimes be included in the output function. It depends on the purpose and behavior of the function itself.

The following function:

\[
\min X \ Y = X, \ X \leq Y
\]

\[
= Y, \text{Otherwise}
\]

takes two arguments and returns either one of them depending on their comparison value of \(X\) and \(Y\). Knowing what that function does, it is noticeable that the first argument can be returned after a function call only
if it is smaller than the second argument. Hence, \( X \) sometimes outlives the function call and will appear in the output. Similar, for the second argument, it can only be included in the output if its value is less than the first argument. Hence, \( Y \) can sometimes outlive the function call and appears in the output function. The Lifetime property information is shown in table 6 using the notation defined above.

<table>
<thead>
<tr>
<th>num -&gt; num -&gt; num</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lifetime Information</td>
</tr>
<tr>
<td>FUNCTION-NAME</td>
</tr>
<tr>
<td>min</td>
</tr>
</tbody>
</table>

**Not-at-all Case:**

The Not-at-all case means that the argument will never be included in the output function. The following function:

\[
Add \ X \ Y = X + Y
\]

takes two arguments as inputs and returns the summation of these two arguments. Analyzing the output behavior of this function, it can be concluded that none of the function arguments will appear in the final result. Thus, the Lifetime property of each argument will not outlive the function call as shown in Table 7.
Table 7 lifetime Information of Add Function

<table>
<thead>
<tr>
<th>FUNCTION-NAME</th>
<th>ARG1</th>
<th>ARG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

List Type:

Lifetime information of a List type argument is more complicated and requires more cases. Lifetime describes whether the List argument can outlive the function call. In addition, it shows how much of the List components can be included in the function output.

Figure 9 List Type Lifetime Information

![Diagram of lifetime information]

Figure 9 explains how lifetime information can be described. It is either the List argument Always, Sometimes, or Not-at-all which outlives the
function call. In a case that does outlive, like in an Always or Sometimes case, it describes how much of the List components can be included in the output after a function call. It can be either All the components in the list, Only Some, or sometimes ALL/Only Some included in the result. Before discussing each case separately, some notations that will be used later are shown in Figure 10 and are defined below.

A:: The list always outlives the function call.

S:: The list sometimes outlives the function call.

N:: The list does not outlive the function call.

Y:: All components in the list will be included in the output

O:: Only some of the components will be included in the output.

B:: All/Only Some of the components will be included in the output.
Always Case:

An Always case describes whether any List argument can always outlive the function call. It also describes how much of that List components can be included in the output, All of them, Only some, or All/Some. Analyzing the behavior of the following functions and what they do, can determine the lifetime property of each one of them:

1. \( tl \ (a:as) \ = \ as \)

2. \( reverse \ \[] \ = \ [] \)

\[
reverse \ (X:XS) \ = \ reverse \ XS \ + \ + \ X
\]

3. \( rduplicate \ \[] \ = \ [] \)

\[
rduplicate \ (a:as) \ = \ a:rduplicate \ as, \ member \ a \ as \ = \ False
\]

\[
\ = \ rduplicate \ as, \ otherwise
\]

Starting with the \( tl \) function that takes a list of any type and returns the tail of that list. All list components for that function will be included in the output except for the first element. Thus, the argument always outlives the function call and Only Some of the list components will be included in the output (lifetime information is shown in Table 8 for such a case).
Table 8  Lifetime Information of Tl, Reverse, and Rduplicate Functions

<table>
<thead>
<tr>
<th>FUNCTION-NAME</th>
<th>ARG</th>
<th>ARG</th>
</tr>
</thead>
<tbody>
<tr>
<td>tl</td>
<td>A</td>
<td>O</td>
</tr>
<tr>
<td>reverse</td>
<td>A</td>
<td>Y</td>
</tr>
<tr>
<td>rduplicate</td>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

Next, *Reverse* function takes one argument of a list type and returns the reverse of that list. Hence, list X always outlives the function call and all components of that list will be included in the output (lifetime information is shown in Table 8 as well).

Finally, *Rduplicate* takes a list argument and returns a list that does not have any duplicates. Hence X always outlives the function call and All/Some of the components might be included in the output (Lifetime information is shown in Table 8).

**Sometimes Case:**

Sometimes case describes whether a list argument sometimes outlives the function call. In a case that it does, it will describe how much of that list component can be included in the result after a function call.

Consider the following function:
Software Retrieval Based on Semantic Properties

1. largest $a \ b = a, \ \text{length } a \geq \text{length } b$
   
   $= b, \ \text{otherwise}$

2. inter $(a : as) \ b = a : \text{inter } as \ b, \ \text{member } a \ b = True$
   
   $= \text{inter } as \ b, \ \text{otherwise}$

3. $F \ a \ b = \text{tl } a, \ \text{hd } a = \text{hd } b$
   
   $= \text{tl } b, \ \text{otherwise}$

Largest function takes two arguments of any list type and returns the list that has more components. The argument $a$ Sometimes outlives the function call. In a case that it does, all the list components are included in the final result. Similar to the first argument, argument $b$ sometimes outlives the function call, all of its components are included in the output. (Lifetime information is shown in Table 9)

**Table 9 Lifetime Information of Largest, Inter, and F Functions**

<table>
<thead>
<tr>
<th></th>
<th>ARG1</th>
<th>ARG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>largest</td>
<td>sometimes</td>
<td>all</td>
</tr>
<tr>
<td>inter</td>
<td>sometimes</td>
<td>all/some</td>
</tr>
<tr>
<td>$F$</td>
<td>sometimes</td>
<td>only some</td>
</tr>
</tbody>
</table>

Inter (intersection) function takes two lists and returns the intersection of these two lists. Depending on the components of both lists, thus, both
arguments sometimes outlive the function call. If they do, either All components or Only Some can be included in the output after a function call. (Lifetime information is shown in Table 9).

$F$ function takes two arguments of any list type and returns the tail of the first argument, if the head of the first argument equals to the head of the second; otherwise it will returns the tail of the second argument. Thus, both arguments sometimes outlive the function call. In a case that they do, only some of the components can be included in the final result after a function call. (Lifetime information is shown in Table 9).

**Not-at-all case:**

Not-at-all cases describe how the argument list does not always outlive the function call and none of the components of that list will be included in the final result.

Consider the following function:

\[
\text{difflist} \ [ ] \ b = []
\]

\[
\text{difflist} (a : as) \ b = a : \text{difflist as} \ b, \ \text{member} \ a \ b = False
\]

\[
= \text{difflist as} \ b, \ \text{otherwise}
\]

\text{difflist} (Difference list) takes two arguments and returns the components that are in the first argument list and not in the second argument list. Thus, the first argument $a$ sometimes outlives the function call and the
All/Some components can be included in the output. On the other hand, the second argument $b$ does not always outlive the function call and none of the components will be included in the result. (Lifetime information shown in Table 10).

<table>
<thead>
<tr>
<th>function-name</th>
<th>ARG1</th>
<th>ARG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>diff-list</td>
<td>S</td>
<td>B</td>
</tr>
</tbody>
</table>

### 2.3 Length Comparison Information

Length comparison is the third property in our research that is used to define a function behavior. It compares the length of the input to the length of the output. Given the following example:

$$tl \ [1, 2, 3, 4] = [2, 3, 4]$$

We can describe the length comparison property of the $tl$ function, which is that the length of the input list is greater than the length of the output list.

WHAT?

Length comparison property is described as follows: "length comparison property shows whether any of the input argument of a function can either
be greater than, less than, or equal to the output."

Figure 11 shows how the length comparison property can be described where:

\(<\): the length of the input is less than the length of the output.

\(\geq\): the length of the input is greater than or equal to the length of the output.

\(\leq\): the length of the input is less or equal to the length of the output.

\(\neq\): the length of the input is equal to the length of the output.

\(\neq\): the length of the input is greater or equal to the length of the output.

\(N/A\): Not applicable, only in the case where the argument does not outlive the function call.

**WHY?**

The importance of the length comparison property is that it helps in defining the behavior of the function. In addition, it helps software retrieval
to distinguish between different functions and enables a retrieval of more accurate and useful candidate components.

Suppose we want to retrieve a function that takes a list and returns the tail of that list. This function is of type \([a] \rightarrow [a]\) and its length comparison property is defined as the length of the input argument which is greater than the length of the output list.

<table>
<thead>
<tr>
<th>[a] (\rightarrow) [a]</th>
<th>Length Comparison Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION-NAME</td>
<td>ARG</td>
</tr>
<tr>
<td>sort</td>
<td>=</td>
</tr>
<tr>
<td>tl</td>
<td>&gt;</td>
</tr>
<tr>
<td>reverse</td>
<td>=</td>
</tr>
</tbody>
</table>

In the software library established, there are three functions that have similar type \([a] \rightarrow [a]\): sort, reverse, and \(tl\) (defined earlier). We know that these functions have length comparison properties which are defined in Table 11. Checking Table 11 for the function that is similar in properties to the function needed. The only one that matches, is the \(tl\) function. Thus, using the Length Comparison property can increase the accuracy and precision of the candidate components.
Define length comparison information:

We can define the length comparison property for a function by analyzing the behavior of that particular function with respect to the input/output relationships that were defined earlier and shown in Figure 11.

The following are some examples to show how length comparison properties can be defined:

$tl$ function, which was defined earlier, takes one argument of type list and returns all components of that list except for the first element. Hence, the length of the input list is greater than the length of the output list. The Length Comparison property is shown in Table 12.

<table>
<thead>
<tr>
<th>[a] -&gt; [a]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length Comparison Information</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FUNCTION-NAME</th>
<th>ARG</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>=</td>
</tr>
<tr>
<td>tl</td>
<td>&gt;</td>
</tr>
<tr>
<td>rduplicate</td>
<td>≥</td>
</tr>
</tbody>
</table>

$sort$ function takes one argument of list type as an input and returns exactly the same list but with sorted components. Thus, the length of the input list equals the length of the output list. The Length Comparison property is also shown in Table 12.
Rduplicate function takes one argument of list type and removes all the duplicate components of that list. The returned list, at most, will have all the input components in the event that there is no duplicate. Thus, the length of the input argument is greater or equal to the length of the output. The Length Comparison property of this function is also shown in Table 12.

Concat function takes two arguments of list type and returns the concatenation of these two lists. The output of this function will have all components of the two input arguments. Thus, the length of each input argument is less than the length of the output argument. The Length comparison property for this is shown in Table 13.

<table>
<thead>
<tr>
<th>[num] -&gt; [num] -&gt; [num]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length Comparison Information</td>
</tr>
<tr>
<td>FUNCTION-NAME</td>
</tr>
<tr>
<td>concat</td>
</tr>
<tr>
<td>condCon</td>
</tr>
</tbody>
</table>

CondCon function takes two input argument lists. If both arguments have the same length then it returns the first argument, otherwise, it returns the concatenation of both argument lists.

CondCon \( a \ b = a, \text{if length } a = \text{length } b \)

\[ = a + + b, \text{otherwise} \]
Thus, in the above example, the length of $A$ is less or equal to the length of output. The length of $B$ is less than the length of the output (in a case where $b$ is included in the output). Length comparison property for this case can be seen as well in Table 13.

2.4 One-to-one Information

One-to-one is another property that helps in defining the behavior of a function. This property defines whether a function is One-to-One or not. This information will be used later in the process of software retrieval to help in the software library $f$ to search for candidate components that meet the user’s requirements.

WHAT?

One-to-one can formally be defined as:

"For general sets $A,B$, If $f: A \rightarrow B$ is one-to-one, then for $a_1, a_2 \in A, f(a_1) = f(a_2) \Rightarrow a_1 = a_2$. consequently, a function of this type can be characterized as one for which each element of $B$ appears at most once as the second component of an order pair in $f"$.[2]

38
Figure 12 shows how one-to-one can be defined where:

- **Y**: the function is one-to-one.
- **N**: The function is not one-to-one

**WHY?**

One-to-one is one of the functions of the semantic property that help to determine the behavior of a function. In addition, it could be used in software retrieval to increase precisicony and retrieve more accurate and useful candidate components.

Suppose we want to retrieve a function that takes one argument of num type and returns the factorial of that value. We know that the factorial function is a One-to-one function. Looking through the software library, we find two functions of similar type num -> num. Each has its own One-to-one
property (as shown in Table 14).

Table 14 One-to-one information of ABS and fact functions

<table>
<thead>
<tr>
<th>num -&gt; num</th>
<th>One-to-one Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION-NAME</td>
<td>ARG</td>
</tr>
<tr>
<td>ABS</td>
<td>N</td>
</tr>
<tr>
<td>FACT</td>
<td>Y</td>
</tr>
</tbody>
</table>

The only one that can match the property of the factorial function is the function Fact. Hence, One-to-one property increases precision and helps in selecting a more accurate component.

**Define one-to-one information:**

One-to-one information can be defined by analyzing the function and the behavior of the input-output relationship. Consider the following examples which are shown in Table 15

Table 15 One-to-one Information of Reverse, Sort, and T1 Functions

<table>
<thead>
<tr>
<th>[a] -&gt; [a]</th>
<th>One-to-one Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNCTION-NAME</td>
<td>ARG</td>
</tr>
<tr>
<td>reverse</td>
<td>Y</td>
</tr>
<tr>
<td>sort</td>
<td>N</td>
</tr>
<tr>
<td>tl</td>
<td>N</td>
</tr>
</tbody>
</table>
The Reverse function takes a list of any list type and returns the reverse of that list. This function always returns a different list for each input list. Hence, the reverse function is One-to-one function.

The Sort function takes a list of any type and returns a sorted list. This function can have two different input lists with the same returned output list as shown in Figure 13. Hence, the sort function is not a One-to-one function.

2.5 Onto Information

Onto is the last property in our research that is used to define the behavior of a function. Onto as a one-to-one property is more formal than the other properties that were defined earlier. But it can be useful in software retrieval by selecting the most candidate components of the user’s defined function.
WHAT?

Onto property can formally be defined as:

"A function \( f: A \rightarrow B \) is called onto, if \( f(A) = B \) (That is, if for all \( b \in B \) there is at least one \( a \in A \) with \( f(a) = b \))."[2]

Figure 14 Onto Information

Figure 14 shows how Onto property can be defined where:

Y:: Function is Onto.

N:: Function is not Onto.

WHY?

The importance of the Onto property is that it can help to determine the behavior of the function. In addition, it can be used in software retrieval to increase precision, and retrieve a more accurate and useful candidate components.
Suppose, we want to retrieve a function that takes one argument of type \( \text{num} \) and returns the absolute value of that argument. We also know that this function is an Onto function. In the software library established, we have two functions of type \( \text{num->num} \) and each one has its own Onto property as shown in Table 16.

<table>
<thead>
<tr>
<th>num -&gt; num</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Onto Information</strong></td>
</tr>
<tr>
<td><strong>FUNCTION-NAME</strong></td>
</tr>
<tr>
<td>ABS</td>
</tr>
<tr>
<td>FACT</td>
</tr>
</tbody>
</table>

The only component that matches the function is ABS function. Thus, the Onto property can help to increase the precision and retrieve the more accurate and useful components.

**Define Onto information:**

The Onto property can be defined by analyzing the function and how the input and output relationship behaves. Consider the following function which are shown in Table 14.

The \text{Fact} function takes one argument of type \( \text{num} \) and returns the factorial of that value. Analyzing this factorial function’s behavior, we find
that some values can not be a result of some factorial's values such as:

\[ ? \]

\[ \text{fact } n = 3 \]

In this case we can not find an input value for \( n \) in order for the result to be 3. Thus, \( \text{Fact} \) function is not an onto. (def. on onto).

\( \text{ABS} \) function takes one argument of type num and returns the absolute of that value. Analyzing \( \text{ABS} \) function, we find that each element of the output domain can have an input. Hence, \( \text{ABS} \) function is an Onto function.
Chapter 3 Retrieval Based on Semantic Properties

A software component can be semantically defined using the semantic properties that were described in the previous chapter. This information tells the user how the function behaves and helps to retrieve more accurate and useful candidate components.

3.1 Retrieval with One Argument

Suppose we want to retrieve a function that takes one argument of type \([a] \rightarrow [a]\) and returns the reverse of that list. (as shown in Figure 2). The behavior of this function can be defined using the semantic properties as so:

1. **Strictness information:**
   - The list is needed.
   - None of the components of the list are needed.

2. **Lifetime information:**
   - The list always outlives the function call.
   - All components of the list will be included in the output.

3. **Length comparison information:**
   - The length of the input is equal to the length of the output.
4. **One-to-one information:**

   - The function is One-to-one.

5. **Onto information:**

   - The function is Onto.

These semantic properties are shown in Table 17.

<table>
<thead>
<tr>
<th>Fname</th>
<th>Strictness Info.</th>
<th>Lifetime Info.</th>
<th>Li-Lo</th>
<th>O-T-O</th>
<th>ONTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>reverse</td>
<td>Y</td>
<td>N</td>
<td>A</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

Using the WISER retrieval process, that is based on the type search key only, will output the following four functions that have similar types as the function needed:

1. reverse :: [a] → [a]
2. sort :: [a] → [a]
3. tl :: [a] → [a]
4. init :: [a] → [a]
In the software library established, these four functions have the semantic properties shown in Table 18.

<table>
<thead>
<tr>
<th>[a] -&gt; [a]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantic properties</td>
</tr>
<tr>
<td>Fname</td>
</tr>
<tr>
<td>reverse</td>
</tr>
<tr>
<td>sort</td>
</tr>
<tr>
<td>tl</td>
</tr>
<tr>
<td>init</td>
</tr>
</tbody>
</table>

Applying the semantic retrieval process, we find that the only function in Table 18 that can match the function wanted is the *Reverse* function. Hence, using semantic properties can increase precision and retrieve more accurate components.

3.2 Retrieval with Two Arguments

Another example that demonstrates a retrieval process, but only using two arguments would occur if we want a function that takes two arguments of a list type and returns the intersection of these two lists (Figure 15).
The semantic properties of this function can be defined as:

**ARG1:**

a. **Strictness information:**
   - □ The list is needed.
   - □ All components of the list are needed.

b. **Lifetime information:**
   - □ The list sometimes outlives the function call.
   - □ Only Some/All components can be included in the output.

c. **Length comparison information:**
   - □ The length of the input greater or equal to the length of the output.

**ARG2:**

a. **Strictness information:**
   - □ The list is needed.
   - □ All components of the list are needed.
b. **Lifetime information:**

- The list sometimes outlives the function call.
- Only Some/All components of the list can be included in the output.

c. **Length comparison information:**

- The length of the input is greater or equal to the length of the output.

**One-to-one information:**

- The function is not One-to-one.

**Onto information:**

- The function is Onto.

These semantic properties are also shown in Table 19 using the notations that were defined in the previous chapter.

---

**Table 19: Semantic Properties of the Intersection**

<table>
<thead>
<tr>
<th>Fname</th>
<th>Semantic properties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Inter</td>
<td></td>
</tr>
<tr>
<td>[a]</td>
<td>-&gt; [a] -&gt; [a]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Semantic properties</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Fname</strong></td>
<td><strong>Strict</strong></td>
</tr>
<tr>
<td>Inter</td>
<td>Y</td>
</tr>
</tbody>
</table>
Again, based on the retrieval process of WISER, we have retrieved the following four functions that have the same type:

1. \( \text{inter} :: [a] \rightarrow [a] \rightarrow [a] \)
2. \( \text{merge} :: [a] \rightarrow [a] \rightarrow [a] \)
3. \( \text{difflist} :: [a] \rightarrow [a] \rightarrow [a] \)
4. \( \text{concatlist} :: [a] \rightarrow [a] \rightarrow [a] \)

In the software library that was established, each one of them is defined by semantic properties and are shown in Table 20.

<table>
<thead>
<tr>
<th>Fname</th>
<th>Strict</th>
<th>Lifetime</th>
<th>L</th>
<th>Strict</th>
<th>Lifetime</th>
<th>L</th>
<th>O-O</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter</td>
<td>Y</td>
<td>A</td>
<td>S</td>
<td>B</td>
<td>≥</td>
<td>Y</td>
<td>A</td>
<td>S</td>
</tr>
<tr>
<td>merge</td>
<td>Y</td>
<td>A</td>
<td>A</td>
<td>Y</td>
<td>&lt;</td>
<td>Y</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>difflist</td>
<td>Y</td>
<td>A</td>
<td>S</td>
<td>B</td>
<td>≥</td>
<td>Y</td>
<td>A</td>
<td>N</td>
</tr>
<tr>
<td>concatlist</td>
<td>Y</td>
<td>N</td>
<td>A</td>
<td>Y</td>
<td>&lt;</td>
<td>Y</td>
<td>N</td>
<td>A</td>
</tr>
</tbody>
</table>

Applying the semantic property retrieval process on Table 20, we found that the only function that can match our function’s semantic properties is the INTER function. Hence using semantic properties in the software retrieval process can retrieve more accurate and useful components.
3.3 Discussion

Ideally, Full Formal Specification is the perfect approach to retrieve the exact desired function from a software library. Because of the difficulty of implementation and usage of a full formal specification, there has been defined a set of semantic properties for the function to help the user to identify the most promising components. This approach is superior to the type signature approach in a way is better in finding useful components than the type signature approach and it is easier. It is not, however, superior to the Full Formal Specification process. The Semantic Properties approach, which is easier than the previously mentioned approaches comes with the cost that sometimes functions can be missed.

A function can be implemented in many different ways, but will always display the same results. The questions of how semantic properties of the same function with different implementation methods can be described and of how semantic properties can miss any components that satisfy the user’s desired function can be answered by discussing each semantic property separately. This is to find out the effectiveness of any missing components.

**Strictness** defined in the previous chapter as the argument value needed in order to evaluate the function result, is somehow related to the implementation of a function. Therefore, Strictness Information can be effective in the way a function is being implemented. For example, suppose we have
the following functions:

\[ F_1 \quad X \quad Y = X, ((X + Y) - Y) > 0 \]

\[ = Y, \text{otherwise} \]

\[ F_2 \quad X \quad Y = X, X > 0 \]

\[ = Y, \text{otherwise} \]

Both Functions \( F_1 \) and \( F_2 \) are the same and are being implemented in two different methods. Strictness information can be described as:

\( F_1 \text{ is strict on both arguments } X \text{ and } Y \)

\( F_2 \text{ is strict only on the first argument } X \)

As it is in the above example, Strictness information for achieving the same function with different implementation methods, also can have different strictness properties.

Similar to the above case, retrieval of components with respect to strictness information can miss some useful components that have different implementation and can not be recoverable.

To solve this problem, the WISER+ provides a way that the user can relax the strictness information of an argument in order to retrieve functions that might match the user’s desired requirements.
On the contrary, **lifetime information** does not depend on the implementation method of a function. It only concerns itself with the result of the function call. This means that the lifetime information would be the same for the same function that has different implementations. In this retrieval process, none of the components that can be useful are missed from the software library being established. In other words, we can say that Lifetime information can recover all the components that satisfy the user's desired requirements.

Lifetime information sometimes retrieves components that have the same lifetime property, but they are different as shown in the following example:

\[
F1 \ a \ b \ c = \ a + b, \ if \ c \\
= \ a * b, \ otherwise
\]

\[
F1 \ a \ b \ c = \ c + 2, \ if \ a \ and \ b \\
= \ c + 1, \ otherwise
\]

Above, both \(F1\) and \(F2\) have the same lifetime properties: none of their arguments can outlive the function call.
Sometimes lifetime property is not enough to distinguish between functions. Using other properties might help to differentiate between them. As in the above example, F1 and F2 differ in their types:

\[\text{F1:}\]

\[a \& b \text{ are of type } \text{num}\]

\[c \text{ is of type } \text{Bool}\]

\[\text{F2:}\]

\[a \& b \text{ are of type } \text{Bool}\]

\[c \text{ is of type } \text{num}\]

Similar to the Lifetime property, \textbf{Length comparison} information does not depend on any implementation methods of a function. It is only concerned with the result of the returned function. Any components that satisfy the user defined requirements will be recovered from the software library that has been established and none of the components will be missed.

\textbf{One-to-one and Onto} properties are more formal than the other three properties. Using these two properties can help to retrieve and recover all components that satisfy the user's defined function. Thus, One-to-one and Onto would not miss any function even if the same function has different implementations, because One-to-one and Onto only deal with the behavior of the function and not the way it is being implemented.
In conclusion, Using semantic properties can recover all the components in a software library except only for the strictness information that sometimes will miss some of the required components. This problem can be solved as we mentioned earlier by relaxing the strictness properties of the components arguments.

Strictness, Lifetime, Length Comparison, One-to-one, and Onto are independent properties. There is no relationship among them. Therefore, the definition of semantic property can not cause any conclusions for other properties.
Chapter 4 WISER+: A Prototype System

WISER+ process is an updated version of WISER that retrieves components based only on the Type search key. WISER+ adds to WISER more features so that it can retrieve components that are more accurate and useful. It makes use of Type as well as Semantic Properties of components in a retrieval approach.

4.1 Overview of WISER

WISER stands for WIndsor SoftwarE for Reuse. It was developed by a Computer Science Graduate Student at the University of Windsor [8]. The purpose of constructing WISER was to build a software database library that could be used to retrieve components based on a type key search. This software database was organized so that the user can be able to retrieve components without searching through all software library's components.
WISER is an automated system that provides many features such as retrieval, insertion, deletion, and browsing as shown in Figure 16.

4.2 WISER+: A New Approach

WISER+ is an automated retrieval system that retrieves candidate components based on the Type and Semantic property (Strictness, Lifetime, Length comparison, One-to-one, Onto) of the components as well. In addition to the retrieval part, WISER+ has the ability to update a library by insertion and
deletion of any component, and to browse through components as shown in WISER+ main menu (Figure 17).

Figure 17 WISER+ Main Menu

4.2.1 Insertion

WISER+ has the ability to insert the type and the five semantic properties of the function in the software library. It makes use of the WISER insertion process to insert the type and on top of that, uses its own insertion process
to insert the semantic properties which are needed as shown in Figure 18.

The insertion works as the user queries the system with insertion of a function that specifies the function’s name, type, and semantic properties. First, the system checks that the software library already established does not have the same function name. If so, the system prompts an error message. Otherwise the type will be inserted by using the WISER insertion method, and the five semantic properties for that particular function will be inserted as well, using semantic property insertion process of WISER+. The five
Semantic properties’s information that will be inserted use the notations that are defined in the previous chapter.

**EXAMPLE:**

Suppose we want to insert a function that takes a list and returns the first element of that list. We specify this function as follows:

1. **Function name:** reverse
2. **Type:** \([a] \rightarrow [a]\)
3. **Strictness information:**
   - □ The list is needed.
   - □ None of the components of the list are needed.
4. **Lifetime information:**
   - □ The list always outlives the function call.
   - □ All of the components will be included in the output.
5. **Length comparison information:**
   - □ The length of the input is equal to the length of the output.
6. **One-to-one information:**
   - □ The function is One-to-one.
7. **Onto information:**
   - □ The function is Onto.
Using the insertion dialog as shown in Figure 19, WISER+ will insert this function with the type and its five semantic properties.

Figure 19 Insertion Dialog (Type & Semantic Properties)

In a case that this function exists, the system will prompt an error message as shown in figure 20.

Figure 20 Error Message of Insertion
4.2.2 Deletion

In the process of updating a software library, the user sometimes encounters the deleting of some components. One of the features of WISER+, is to handle this deletion problem as shown in Figure 21. WISER+ deletion process deletes the function name as well as the type and the five semantic properties of that function from the software library.

![Figure 21: Deletion Process](image)
The deletion part works as the user queries the system by using the function name and type. First, the system checks for the availability of the function in the database. In case it is not available, the system prompts an error message. Otherwise, it automatically deletes the function name from the type that was specified by using the WISER deletion approach. Then it deletes the five semantic properties of that particular function with the new WISER+ deletion approach.

**EXAMPLE:**

WISER+ offers a method to delete a function from a software library. Thus, the user can accomplish this by using a deletion dialog as shown in Figure 22.

Figure 22 Deletion Dialog
The user only has to specify the name and the type of the function need to be deleted. WISER+ will then delete the function name, type, and semantic properties from that specific software library.

4.2.3 Retrieval

The main purpose of the WISER+ system is to retrieve candidate components that satisfy the user’s desired requirements. WISER+ offers a new retrieval approach so as to retrieve more accurate and useful components. As shown in Figure 23, WISER+ first uses WISER retrieval approach to retrieve candidate components based on the type search key. Then, it applies the five semantic properties process method to retrieve only the components that match the user specified properties.
WISER+ retrieval works as the user queries the system with the type, number of arguments, and the five semantic properties of such a function. The user has the choice of either applying the type retrieval implicitly or explicitly.

In the implicit case, the system will prompt the candidate components that satisfy the type and five semantic properties, either with an exact, general, specific, or freezing argument match.

On the other hand, in the case of explicitness, the system first prompts the candidate components that satisfy the type key only. It leaves a choice for the user to apply the semantic retrieval process either on an exact, general, specific, or freezing match.

In case there is no match, the system will prompt a message telling the user of the unavailability of any candidate components that satisfy the user’s requirements.

**EXAMPLE:**

Suppose we want to retrieve a function that takes a list of any type and returns the tail of that list. The five semantic properties of this function are defined as:

1. **Type:** [a] → [a]
2. **Strictness information:**

   □ The list is needed.
3. **Lifetime information:**

- The list always outlives the function call.
- Only some of the components will be included in the output.

4. **Length comparison information:**

- The length of the input is greater than the length of the output.

5. **One-to-one information:**

- The function is not One-to-one.

6. **Onto information:**

- The function is Onto.

Then, a query is made to WISER+, either with the type first or with the type and five semantic properties as shown and explained in Figure 23.

In this case, we are going to start querying with the type search key first and then apply the semantic properties retrieval process.
Upon the success of the query with the type search (Figure 24), WISER+ returns exact matching functions which appear in Figure 25. The user now has the choice to apply the semantic property retrieval process on these functions.
In case the user chooses to apply semantic properties process, Semantic dialog as shown in Figure 26 will appear. This will result in retrieving candidate functions that satisfy the type and five semantic properties as well (Figure 27).
Software Retrieval Based on Semantic Properties

Figure 26 Semantic Properties Retrieval Dialog

Figure 27 Exact Match (Type & Semantic Properties)
The $tl$ function is one of the functions that is retrieved with respect to the type and five semantic properties.

The user also has the choice to look for some functions that have general, specific or freezing arguments as shown in Figure 28, Figure 29, & Figure 30.

Figure 28 General Match (Type & Semantic Properties)
Figure 29 Freezing Match (Type & Semantic Properties)

Figure 30 Specific Match (Type & Semantic Properties)
4.2.4 Browser

Sometimes the user may feel the need to inspect the five semantic properties for a function in the already established repository library. WISER+ provides a tool which allows the user to search through the library for semantic properties of the stored components.

Figure 31 Browser Process

The WISER+ browser is implemented in a way that allows a user to inspect semantic properties of the stored components, as shown in Figure 31. The user has the choice to browse through functions either by their
argument numbers, exact match, general match, specific match, or freeze match of their argument type. Depending on the user’s choice, a function can be inspected with its five semantic properties. In addition, users are allowed to inspect different functions by making different choices after they are through browsing with their previous options.

EXAMPLE

Suppose we want to browse functions that have two arguments. This choice results in functions that appear in the upper section of Figure 32.
Clicking on the function *diffList*, for example, results in the appearance of the five semantic properties of that particular function in the lower section of the browser window (Figure 33). Also, we can inspect other functions by clicking again on a function name which is on the upper section of a browser window.

Figure 33  Semantic Properties of DiffList (Shown in the Lower Section in Browser Window)
Chapter 5 Applications

WISER+ was developed for the purpose of helping a user to retrieve components that can be useful in building new software applications. In this chapter, two applications will be discussed that are used in the application on type search key[8]. The reason for using the same examples is to show how the retrieval based on semantic properties (defined earlier) can increase the precision of the candidate components.

5.1 Prime Number

The first application is prime number function. The purpose of this function is to generate a list of prime numbers up to a specific integer n.

Etathosthoues proposed a method to generate a list of prime numbers. His method worked as follows: first, a list of sequence integers is generated, starting from 2 and building up to specific integer n. Then, taking the first element of that generated list as a prime number and deleting all the elements that are divisible by that number, and finally repeating the same procedure for the rest of the resulted list, until it is empty.

Table 21 Primelist Miranda Program

<table>
<thead>
<tr>
<th>Primelist n = sieve (Genlist 2 n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>seive [] = []</td>
</tr>
<tr>
<td>seive (a:as) = a:seive [i</td>
</tr>
</tbody>
</table>
Table 21, [8], shows the Miranda program for *PrimeList*. This program reuses some of Miranda components that are stored in a repository library of the WISER+.

*Genlist* takes two arguments of type num and returns a list of num starting with the value of the first argument (2 in this case) up to the value of the second argument.

The semantic properties of this function are:

**Type:** num -> num -> [num]

**ARG1:**

The argument is needed.

The argument outlives the function call.

The length of input is less than the length of the output.

**ARG2:**

The argument is needed.

The argument outlives the function call.

The length of input is less than the length of the output.

The function is One-to-one and Onto.

Querying WISER with type *num* --> *num* --> *num*, the system will result in the following function manner:

Genlist n m = [n..m]
Querying the WISER+ system with the above semantic properties, the system will yield the following function:

\[ \operatorname{Genlist} n \ m = [n..m] \]

Inspecting the source code of the above functions, we found that it is the function that we need for our application with only one modification. This modification is to change the first argument from a non-fixed value to a fixed value 2.

\[ \operatorname{Genlist} 2 \ m = [2..m] \]

The other function that is reused, is the append function (\(\cdot\)). This function takes two arguments and returns a list that appends the first argument to the list of the second argument. The five semantic properties of this function are as follow: Type: \( a \rightarrow [a] \rightarrow [a] \)

ARG1:

The argument is needed.

The argument outlives the function call.

Length of the input is less than the length of the output.

ARG2:

The argument is needed.

None of the components are needed.
The argument outlives the function call.

All the components of the list will be included in the output.

Length of the input is less than the length of the output.

The function is One-to-one and Onto.

Querying WISER with type $\text{num} \rightarrow \text{num} \rightarrow \text{num}$, the system will result with the following:

1. postfix
2. insert
3. eliminate
4. concat

Querying WISER+ system with the above semantic properties will result in obtaining the following function:

1. concat

Inspecting the source code of the above functions, we found out that concat (:) function suits our application.

5.2 Magic Square Problem

The Magic square problem is another application that reuses Miranda functions from the WISER+ repository library. The Magic square problem
is to build a $m \times m$ matrix of integers from 1 to $m^2$, where $m$ is odd. This matrix is filled with integers in a way that the sum of each row, column, and diagonal has the same value. In the case of $m = 5$, as shown in Table 22, the sum of each row, column, and diagonal has the same value: 65.

Table 22 Magic Square Matrix

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>24</td>
<td>1</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>23</td>
<td>5</td>
<td>7</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>13</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>19</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>18</td>
<td>25</td>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>

Coxter suggested a method to solve the magic square problem. He states that "start with 1 in the middle of the top row; then go up and right assigning numbers in increasing order to the empty squares; if you fall off the square, imaging the same square is tiling the plane and continue; If a square is occupied, move down instead and continue."
Table 23 Magic square Miranda Program

```plaintext
magic m = error "Dimension of Magic Table Must Be Odd", (m mod 2) = 0
    = "[1]", m = 1
    = print([L|x<-[1..m];L<-({getline x (sortList(makeList m))})));, otherwise

where

getLine x ax = [[i,(i,j) <-{sortTuple [{(n,z)|(n,z,y)<-ax; y=x)}]]

SortTuple [] = []

sortTuple ((a,c):xs) = sortTuple [(k,m)|(k,m)<-xs; m<=c]
    ++ [(a,c)]
    ++ sortTuple [(k,m)|(k,m)<-xs; m>c]

sortList [] = []

sortList ((a,b,c):xs) = sortList [(k,m)|(k,m)<-xs; m<=c]
    ++ [(a,c)]
    ++ sortList [(k,m)|(k,m)<-xs; m>c]

makeList m = theList (m, m^2-1)

case (a,b)
    next [{b(a,div2)+1,1}], if b = 1
    next (theList (a,b-1)), otherwise

where

next [z]  = (coor z): [z]

next (z:zs) = (coor z): z:zs

coor (n,x,y) = (n+1,x,y+1), (n mod a) = 0
    = (n+1,x+1,a), y=1
    = (n+1,x+1,y-1), x=a
    = (n+1,x+1,y-1), otherwise

print [] = []

print (x:xs) = show x ++ "\n" ++ print xs
```
Based on the above method, Table 23 [8] shows Miranda program solution of a magic problem. This program reuses number from Miranda functions that are taken from the WISER+ repository library.

Concat (++) function takes two arguments of list type and returns the concatenation of both arguments. This function has semantic properties as follows:

Type: [a] -> [a] -> [a]

ARG1:

The list is needed but none of the components are needed.
The list outlives the function call. Also, all the components will be included in the result.
The length of input list is less than the length of the output.

ARG2:

The list is needed but none of the components are needed
The list outlives the function call. Also, all the components will be included in the result.
The length of input list is less than the length of the output.

The function is not One-to-one but it is Onto.

When querying WISER with type [a] --> [a] --> [a], the system results as follows:

1. merge
2. concatlist

3. difflist

4. inter

On the other hand, querying WISER+ with the above semantic properties results in by obtaining the following:

1. concatlist :: [a] → [a] → [a]

Inspecting the source code of the function, Concatlist(++) is the function that suits our needs.

In this program, we need also a sort function that can be semantically defined as:

- type: [a] → [a]

ARG:

The list is needed.

None of the components of the list are needed.

The argument list always outlives the function call.

All the components of the list will be included in the output.

Length of the input is equal to the length of the output.

The function is not One-to-one but it is Onto.
Querying WISER with type \([\text{a}] \rightarrow \text{[a]}\), the system results in the following:

1. reverse
2. mkset
3. sort
4. elizero
5. tl
6. init

Querying WISER+ with the above information, results in obtaining the following:

1. sort :: \([\text{a}] \rightarrow \text{[a]}\)

Inspecting the source code of the resulting function, \textit{Sort} is the function that suits our needs with little modification.

Finally, to print out the result, we need a function that takes one argument of any type and returns a print out of an argument which is of type \([\text{char}]\). This function can be semantically defined as:

type \(a \rightarrow \text{[char]}\)

ARG:

The argument is needed.

The argument does not outlives the function call.

Length of the input is equal to the length of the output.
The function is One-to-one and Onto.

Querying WISER+ with the above information, results in obtaining the show function which is the one that suits our particular needs.

5.3 Comparison of WISER & WISER+

In the above two applications, WISER+ is able to identify a more accurate and useful components than WISER.

WISER+ eliminated components that were not useful for our applications. In the case of the sort function, WISER retrieved the following function:

1. tl :: [a] → [a]
2. reverse :: [a] → [a]
3. sort :: [a] → [a]
4. init :: [a] → [a]
5. mkset :: [a] → [a]
6. elizero :: [a] → [a]

On the other hand, WISER+ retrieved only sort function, and eliminated functions that were not useful and that did not meet any specifications.

Therefore WISER+ increases the precision of a query and helps the user in retrieving more promising candidate components.
Chapter 6 Conclusion & Future Work

6.1 Conclusion

The work of this thesis investigated a number of properties of a function that are useful for storing/retrieving reusable components with respect to a user’s requirements.

A number of semantic properties of function were identified such as: Strictness, Lifetime, Length Comparison, One-to-one, and Onto. The information from these five semantic properties were useful in WISER+ to identify accurate and useful candidate components.

Using Types along with Semantic properties resulted in the following benefits:

- Increased the recall and precision of a query.
- Easier to use than full formal specification.
- No need for expertise on full formal specification.
- Applying semantic properties retrieval on top of a type search key results in more useful candidate components than the type search key only.
- WISER+ provided a useful automatic retrieval process as shown in chapter 4.
- WISER+ provided a browser system. The user could browse through the Semantic properties of a function.
☐ Easiest to use. WISER+ provided an interface menu that made it easier for the user.

6.2 Future Work

Software retrieval based on semantic properties is more successful for retrieval of more accurate and useful components than the type search key. Despite all the benefits, sometimes WISER+ retrieves components that do not suit the user’s requirements. Therefore, additional techniques or methods are needed to assure retrieval candidate components which must meet the user’s function requirements. In the sense of retrieving more appropriate components, the result from a retrieval process using both type and semantic properties may be executed/tested on a set of input data to achieve more promising components.
BIBLIOGRAPHY


VITA AUCTORIS

Khaled Khalil was born in 1964 in Beruit Lebanon. He graduated from Ras-Nabaa High School in 1985. From there he went on to the University of Windsor where he obtained a B. Sc. in Computer Science in 1993. He is currently a candidate for the Master’s degree in Computer Science at the University of Windsor and hopes to graduate in the Fall of 1995.