Execution based retrieval of reusable software components.

Ping Bai

University of Windsor

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

Recommended Citation

This online database contains the full-text of PhD dissertations and Masters’ theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.
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.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.
Execution Based Retrieval of Reusable Software Components

by
Ping Bai

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 Computer 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.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-10972-0
### THE HUMANITIES AND SOCIAL SCIENCES

#### Communication and the Arts
- Architecture: 0729
- Art History: 0777
- Cinema: 0790
- Dance: 0728
- Fine Arts: 0727
- Information Science: 0109
- Journalism: 0191
- Library Science: 0599
- Mass Communications: 0708
- Music: 0411
- Speech Communication: 0429
- Theater: 0465

#### Education
- General: 0515
- Administration: 0514
- Adult Education: 0516
- Agricultural: 0517
- Art: 0773
- Bilingual and Multicultural: 0282
- Business: 0588
- Curriculum and Instruction: 0527
- Elementary: 0518
- Elementary Education: 0522
- Guidance and Counseling: 0528
- Health: 0519
- Higher: 0525
- History: 0520
- Home Economics: 0278
- Industrial: 0521
- Language and Literature: 0199
- Mathematics: 0280
- Music: 0299
- Philosophy: 0206
- Physical: 0523

#### Psychology, Religion and Theology
- Psychology: 0126
- Religion: 0135
- Sociology: 0146
- Special: 0129
- Teacher Education: 0130
- Technology: 0710
- Text and Measurements: 0288
- Vocational: 0747

#### Language, Literature and Linguistics
- General: 0679
- American: 0673
- Canadian: 0680
- Canadian (English): 0682
- Canadian (French): 0683
- Comparative: 0693
- German: 0611
- History: 0678
- Italian: 0672
- Latvian: 0673
- Middle Eastern: 0684
- Romance: 0683
- Slavic and East European: 0681

#### Philosophy, Religion and Theology
- Philosophy: 0422
- Religious Studies: 0421
- Theology: 0469

#### Social Sciences
- American Studies: 0323
- Anthropology: 0294
- Cultural: 0296
- Physical: 0327

#### Economic Sciences
- General: 0501
- Agricultural: 0502
- Business: 0503
- Management: 0504
- Marketing: 0505
- Canadian: 0506

#### Medical Sciences
- General: 0133
- History: 0134
- Sociology: 0135

#### Political Sciences
- General: 0615
- International Law: 0614
- Public Administration: 0617

#### Physical Sciences
- General: 0524
- Nuclear: 0525
- Physical: 0526

#### The Sciences and Engineering

#### Biological Sciences
- General: 0473
- Agronomy: 0268
- Animal Culture and: 0475
- Animal Pathology: 0476
- Food Science and: 0359
- Forestry and Wildlife: 0478
- Plant Culture: 0489
- Plant Physiology: 0498
- Wood Technology: 0746

#### Health and Environmental Sciences
- Environmental Sciences: 0761
- Health Sciences: 0566

#### Physical Sciences
- Pure Sciences: 0482
- General: 0483

#### Applied Sciences
- Engineering: 0537
- Aerospace: 0458
- Agricultural: 0529
- Atmospheric: 0540
- Biomedical: 0541
- Chemical: 0542
- Electrical: 0543
- Heat and Thermodynamics: 0544
- Industry: 0545
- Industrial: 0546
- Marine: 0547
- Mechanical: 0548
- Metallurgy: 0549
- Nuclear: 0550
- Packaging: 0551
- Petroleum: 0552
- Sanitary and Municipal: 0553
- Geological: 0423
- Operations Research: 0554
- Robotics: 0426
- Textile Technology: 0494

#### Psychology
- General: 0421
- Behavioral: 0422
- Clinical: 0420
- Developmental: 0420
- Experimental: 0421
- Industrial: 0422
- Personality: 0425
- Psychological: 0426
- Psychometrics: 0427
- Social: 0428
Abstract

Software reuse is widely believed to be a promising means for improving software productivity and reliability. However, reuse is only effective if it is easier to locate/retrieve (and appropriately modify) a reusable component than to write it from scratch. Searching for effective method for retrieving software components becomes increasingly important. In this thesis, we explore a practical method to retrieve software components by executing them on test data generated by the system and determining the components to be retrieved by verifying the results. A prototype system was designed and developed based on this approach.
To my parents
my husband Shudao
my children Michael and Grace
Acknowledgments

I would like to acknowledge the support and guidance provided by Dr. Y. Park, whose time, patience and effort have contributed in guiding me through my study and this thesis work. Without his help and guidance, I would not have finished my work in this period of time. I am grateful to Dr. R. Frost for providing valuable suggestions and comments that have aided in my progress and thanks to Dr. K.Y. Fung for being willing to serve as external reader.

A special thanks to my husband for always being there for me with his support and understanding. Finally I would like to thank all my colleagues in the Graduate Lab in The School of Computer Science at the University of Windsor.
Table of Contents

Abstract ................................................................. iv
Acknowledgments ......................................................... vi
List of Figures .......................................................... x
List of Tables ........................................................... xiii

1 Introduction .......................................................... 1
   1.1 Software Reuse .................................................. 1
   1.2 Functional Programming Environment ......................... 3
       1.2.1 Polymorphic Functions ................................... 3
       1.2.2 Higher-Order Functions .................................. 5
       1.2.3 Recursion and Induction .................................. 5
   1.3 An Overview of WISER ......................................... 7
   1.4 Overview of the thesis ......................................... 10
       1.4.1 Motivation ................................................ 10
       1.4.2 The Objectives of the Thesis Work ....................... 10
       1.4.3 Thesis Statement ......................................... 10
       1.4.4 Organization of the Thesis ............................... 11

2 An Approach to Execution-based Retrieval ....................... 12
   2.1 The Process of Execution Based Retrieval .................. 12
   2.2 The Selection of Candidate Functions ....................... 13
   2.3 The Strategy of Test Data Generation ....................... 14
       2.3.1 Test Cases for Basic Types ............................. 15
           2.3.1.1 Number Type ..................................... 15
           2.3.1.2 Boolean Type ....................................... 16
2.3.1.3 Character Type ..................... 16
2.3.1.4 List Type .......................... 16
2.3.1.5 Tuple Type .......................... 18
2.3.2 Test cases for Polymorphic Types ............... 20
2.3.3 Test cases for the Functions with more than one
Arguments .................................. 21
2.3.4 Test Cases for Higher-Order Functions .......... 22
2.4 Execution and Verification ........................ 24
3 A Prototype System ................................ 27
3.1 System Design ................................ 27
3.2 Implementation ................................ 28
  3.2.1 Retrieval ................................ 29
    3.2.1.1 The Selection of Candidate Components ... 30
    3.2.1.2 Test Data Generation .................... 31
    3.2.1.3 Execution and Run-time Exception Handling . 35
    3.2.1.4 Verification of Results .................. 37
3.2.2 Insertion ................................ 37
3.2.3 Deletion .................................. 38
3.2.4 Saving and Browsing .......................... 38
3.3 User Interface and Examples .................. 39
  3.3.1 Examples of Retrieval ...................... 42
    3.3.1.1 Retrieving function sort ................ 42
    3.3.1.2 Retrieving function filter ............... 47
  3.3.2 Insertion and Deletion ...................... 50
3.3.3 Browser ............................... 52
4 Linking the Retrieval System to the World Wide Web ............... 54
   4.1 Internet and the World Wide Web ......................... 54
   4.2 Fill-out Form and Common Gateway Interface (CGI) .......... 57
   4.3 Running Retrieval System Through CGI .................... 58
   4.4 An Example of Retrieval Using the Web Browser .......... 60
5 Conclusions and Future Work ........................................ 66
   5.1 Conclusions ...................................................... 66
   5.2 Future Work ...................................................... 68
BIBLIOGRAPHY ................................................................. 69
APPENDIX A WISER Web Pages ............................................. 71
   A.1 WISER Home Page ............................................... 71
   A.2 About-WISER Page ................................................ 72
   A.3 WISER Browsing Page ............................................. 73
   A.4 WISER Retrieval Page ............................................ 74
   A.5 Insertion Page ..................................................... 75
   A.6 Type Retrieval Page (Exact Match Functions) ............... 76
   A.7 General Functions (Type retrieval) ........................... 77
   A.8 Freezing Argument Functions (Type retrieval) ............... 78
   A.9 Semantics Based Retrieval Page ................................. 79
   A.10 Execution Based Retrieval Page ................................. 80
   A.11 User Testing Page ................................................ 81
   A.12 System Testing Page .............................................. 82
   A.13 Execution Based Retrieval Result Page ....................... 83
APPENDIX B VITA AUCTORIS ............................................... 84
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>An example of higher-order function</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Natural number induction</td>
<td>7</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Lists induction</td>
<td>7</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Query example 1.</td>
<td>9</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Query example 2.</td>
<td>9</td>
</tr>
<tr>
<td>Figure 6</td>
<td>The process of Execution-based Retrieval</td>
<td>12</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Selection of candidate components</td>
<td>14</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Test cases for $\text{num}$ type</td>
<td>15</td>
</tr>
<tr>
<td>Figure 9</td>
<td>An example of test cases for $\text{num}$ type</td>
<td>15</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Test cases for $\text{bool}$ type</td>
<td>16</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Test cases for $\text{char}$ type</td>
<td>16</td>
</tr>
<tr>
<td>Figure 12</td>
<td>An example of test cases for $\text{num}$ list</td>
<td>18</td>
</tr>
<tr>
<td>Figure 13</td>
<td>An example of test cases for $\text{bool}$ list</td>
<td>18</td>
</tr>
<tr>
<td>Figure 14</td>
<td>An example of verification of results</td>
<td>26</td>
</tr>
<tr>
<td>Figure 15</td>
<td>An overview of the system design</td>
<td>29</td>
</tr>
<tr>
<td>Figure 16</td>
<td>The design of execution-based retrieval</td>
<td>29</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Retrieval based on types</td>
<td>30</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Test data generation</td>
<td>32</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Examples of test case for basic data type and tuple type</td>
<td>33</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Data flow of test data generation for first argument of query $[[\text{num}]] \rightarrow \text{nnm}$</td>
<td>33</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Test data for query $[[\text{num}]] \rightarrow \text{num}$</td>
<td>34</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Validation check</td>
<td>36</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Example 1 Validation on test data</td>
<td>36</td>
</tr>
<tr>
<td>Figure 24</td>
<td>An example of verification of results</td>
<td>37</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Insertion</td>
<td>38</td>
</tr>
<tr>
<td>Figure 26</td>
<td>Deletion</td>
<td>39</td>
</tr>
<tr>
<td>Figure 27</td>
<td>Organization of windows</td>
<td>40</td>
</tr>
<tr>
<td>Figure 28</td>
<td>Main Window</td>
<td>41</td>
</tr>
<tr>
<td>Figure 29</td>
<td>Exe-Retrieve Window</td>
<td>43</td>
</tr>
<tr>
<td>Figure 30</td>
<td>Candidates Window</td>
<td>44</td>
</tr>
<tr>
<td>Figure 31</td>
<td>System Test Window</td>
<td>45</td>
</tr>
<tr>
<td>Figure 32</td>
<td>Match Functions Window</td>
<td>46</td>
</tr>
<tr>
<td>Figure 33</td>
<td>Saving Window</td>
<td>46</td>
</tr>
<tr>
<td>Figure 34</td>
<td>User Test Window</td>
<td>47</td>
</tr>
<tr>
<td>Figure 35</td>
<td>Retrieval Window for query (a → bool) → [a] → [a]</td>
<td>48</td>
</tr>
<tr>
<td>Figure 36</td>
<td>Candidates Window for query (a → bool) → [a] → [a]</td>
<td>49</td>
</tr>
<tr>
<td>Figure 37</td>
<td>System Test Window for query (a → bool) → [a] → [a]</td>
<td>50</td>
</tr>
<tr>
<td>Figure 38</td>
<td>Match Functions Window for query (a → bool) → [a] → [a]</td>
<td>50</td>
</tr>
<tr>
<td>Figure 39</td>
<td>Insertion Window</td>
<td>51</td>
</tr>
<tr>
<td>Figure 40</td>
<td>Deletion Window</td>
<td>52</td>
</tr>
<tr>
<td>Figure 41</td>
<td>Source Code Window</td>
<td>53</td>
</tr>
<tr>
<td>Figure 42</td>
<td>Browser Window</td>
<td>53</td>
</tr>
<tr>
<td>Figure 43</td>
<td>A client/server model for data communication</td>
<td>54</td>
</tr>
<tr>
<td>Figure 44</td>
<td>The Web within the Internet</td>
<td>55</td>
</tr>
<tr>
<td>Figure 45</td>
<td>Hypertext documents</td>
<td>56</td>
</tr>
<tr>
<td>Figure 46</td>
<td>A URL diagram</td>
<td>57</td>
</tr>
</tbody>
</table>
Figure 47  A schematic overview of data flow using CGl. ......... 58
Figure 48  Organization of Web pages for WISER ............... 59
Figure 49  WISER Home Page .................................. 62
Figure 50  WISER Retrieval Page ............................... 63
Figure 51  WISER System Testing page ......................... 63
Figure 52  WISER Execution Based Retrieval Page .............. 64
Figure 53  WISER User Testing Page ............................ 65
Figure 54  WISER Execution Based Retrieval Results Page .... 65
Figure 55  WISER Home Page .................................. 71
Figure 56  WISER Browsing Page ............................... 73
Figure 57  WISER Retrieval Page ............................... 74
Figure 58  WISER Insertion Page ............................... 75
Figure 59  WISER Retrieval Based on Type (Exact Match Components) ............................................. 76
Figure 60  WISER General Match Components Pages .......... 77
Figure 61  WISER Freezing Arguments Components Page ...... 78
Figure 62  WISER Semantics Properties Based Retrieval Page . 79
Figure 63  WISER Execution Based Retrieval Page .............. 80
Figure 64  WISER User Testing Page ............................ 81
Figure 65  WISER System Testing Page .......................... 82
Figure 66  WISER Execution Based Retrieval Result Pages .... 83
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Type constructors</td>
<td>4</td>
</tr>
<tr>
<td>Table 2</td>
<td>Record structure</td>
<td>8</td>
</tr>
<tr>
<td>Table 3</td>
<td>Examples of list value</td>
<td>16</td>
</tr>
<tr>
<td>Table 4</td>
<td>Examples of list type</td>
<td>17</td>
</tr>
<tr>
<td>Table 5</td>
<td>Test cases for list type</td>
<td>17</td>
</tr>
<tr>
<td>Table 6</td>
<td>Test cases for each component of a tuple pair ((\text{num}, \text{bool}))</td>
<td>19</td>
</tr>
<tr>
<td>Table 7</td>
<td>The final test cases for the query ((\text{num}, \text{bool}) \rightarrow \text{num})</td>
<td>19</td>
</tr>
<tr>
<td>Table 8</td>
<td>An example of test data generation for more complex data type ([\text{[num]}] \rightarrow \text{num})</td>
<td>19</td>
</tr>
<tr>
<td>Table 9</td>
<td>Test data generation for query ([\text{a}] \rightarrow [\text{a}])</td>
<td>20</td>
</tr>
<tr>
<td>Table 10</td>
<td>Test data for each argument for the query (\text{num} \rightarrow \text{bool} \rightarrow \text{num})</td>
<td>21</td>
</tr>
<tr>
<td>Table 11</td>
<td>Test cases for each argument of the function</td>
<td>21</td>
</tr>
<tr>
<td>Table 12</td>
<td>Test cases for the components with more than one arguments</td>
<td>22</td>
</tr>
<tr>
<td>Table 13</td>
<td>Number of test cases for basic type</td>
<td>22</td>
</tr>
<tr>
<td>Table 14</td>
<td>Test cases for each argument</td>
<td>23</td>
</tr>
<tr>
<td>Table 15</td>
<td>Test cases for query: ((\text{num} \rightarrow \text{bool}) \rightarrow [\text{num}] \rightarrow [\text{num}])</td>
<td>24</td>
</tr>
<tr>
<td>Table 16</td>
<td>Test cases and the corresponding results for retrieving function \text{filter}</td>
<td>24</td>
</tr>
<tr>
<td>Table 17</td>
<td>Base cases for each basic data type</td>
<td>31</td>
</tr>
</tbody>
</table>
Table 18 | The algorithm of test data generation | 32
---|---|---
Table 19 | Test cases of each argument for the query $[\text{num}] \rightarrow \text{num} \rightarrow \text{num}$ | 34
Table 20 | Test data for the query $[\text{num}] \rightarrow \text{num} \rightarrow \text{num}$ | 35
Table 21 | Test data generated by system and result values from user for query $[\text{num}] \rightarrow [\text{num}]$ | 45
Table 22 | Test cases and the corresponding results for retrieving function $\text{filter}$ | 49
1.1 Software Reuse

The idea of formal software reuse, as first introduced by McIlroy in his 1968 seminal paper, entailed the development of an industry of reusable source-code software components and the industrialization of the production of application software from off-the-shelf components [9]. Software reuse is now understood to encompass all the resources used and produced during the development of software and widely believed to be a promising means for improving software productivity and reliability.

"Software reuse is the process of using existing software artifacts rather than building them from scratch" [12]. The primary motivation to reuse software artifacts is to reduce the time and effort required to build software systems. The quality of software systems is enhanced by reusing quality software artifacts, which also reduces the time and effort required to maintain software systems.

The process of software reuse can be divided into three phases: classification/storage phase, which deals with how the components are classified and stored in a software library, location/retrieval phase, which deals with how the components are located/retrieved from a large collection and the final phase which deals with adaptation/integration of the components after they are found in a software library.

As the demand for cost-effective software rises, reuse becomes increasingly important as a potential solution to the problem of low programmer productivity. A fundamental problem in software reuse is the lack of tools to locate potential codes for reuse. The search for effective methods of promoting software reuse has a strong economic basis. Many approaches to automatic retrieval of reusable software components have been
Introduction

proposed and implemented. Automated retrieval of software components can be divided into two main approaches. One approach is termed the syntactic approach, which deals with the syntax of the components. Keyword and faceted methods can be categorized under this approach. Using keywords to aid in location/retrieval requires users to have knowledge of both the keyword set and software library. The faceted method is quite similar to the library classification of components based on the subject. By this method, subject statements are analyzed into their components’ elemental classes, and these classes are listed in the schedule. Another approach is called the semantic approach, which deals with the meaning of components. Under this approach there are several techniques for retrieval. The first is retrieval based on the types of components, which uses component types as search keys. The basic idea of this technique is that retrieval is achieved by matching components’ type to the query. Although this technique is less accurate, it gives a rough cut to the library components. The second retrieval technique is based on specification. Formal specification is an ideal way to locate/retrieve components. Unfortunately it is very difficult to implement and difficult for both user and system developer because of the full knowledge of the specification language. We believe that formal specification-based matching will remain a theoretical curiosity. However, if the domain is restricted, then it might be possible to decide whether the components are equivalent. So there is another technique based on specification, which we call “semantic-properties-based retrieval” which uses some semantic properties of components as search keys to achieve retrieval. Finally we have a method called “execution-based retrieval”, which deals with the behavior of executable components. The basic idea of execution-based retrieval is that the system executes each component on the test data provided either by the user or generated by the system and then determines the components to be retrieved by comparing the results with the known outputs. This method exploits the
executable properties of software and the retrieval results are much closer to the behavior of the components required by the user. Two ideas were proposed: Sampling Behavior [14] and Generalized behavior-based retrieval [10]. The basic idea behind sampling behavior is that the system generates test data randomly from the domain of argument and then completes testing to achieve retrieval. For generalized behavior-based retrieval, test data come from the user. One problem in existing execution-based retrieval is that the test data set may be large due to the random generation. It may take a long time to complete the testing. Another problem is that random sampling needlessly reduces the precision of retrieval in some cases.

1.2 Functional Programming Environment

"Programming in a functional language consists of building definitions and using the computer to evaluate expressions. Programmers in such a language are primarily responsible for the construction of a function that will ultimately solve the problem at hand" [15].

A functional program usually consists of a series of function definitions followed by an expression that involves application of the functions. Functions can be passed as arguments to other functions, returned by functions as results. Polymorphic functions allow the same functions to be applied to different data types. These features make functional languages very higher-level programming languages. They can be used to produce a program more quickly than is possible with imperative language and their lack of side effects, due to the omission of assignment operator, means that they are more amenable to mathematical reasoning.

Polymorphic Functions Functional languages like Miranda are polymorphic strong typing languages. This means that each expression or subexpression has a type which
can be deduced at compile time. Types can be divided into two groups: primitive types and compound types whose values are constructed from those of other types.

\textit{num, bool, char} are well-known primitive data types in functional languages. Three forms of type constructors for more complex type are provided by functional languages: (see Table 1).

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha \rightarrow \beta$</td>
<td>Function of argument type $\alpha$ and result $\beta$</td>
</tr>
<tr>
<td>$\alpha \times \beta$</td>
<td>Pair with left and right comps. of type $\alpha$ and $\beta$</td>
</tr>
<tr>
<td>$[\alpha]$</td>
<td>List containing zero or more elements each of type $\alpha$</td>
</tr>
</tbody>
</table>

Table 1 Type constructors

Example 1: \( F :: \quad \text{num} \rightarrow [\text{num}] \rightarrow \text{num} \)

Example 2: \( \text{lenlist} :: \quad [a] \rightarrow \text{num} \)

Example 1 is a function type which takes two arguments. The first one has type \text{num}, the second one has type \([\text{num}]\) which is a number list, and the result type is \text{num}. In example 2, the function has one argument with list type \([a]\) and the result type is \text{num}. Here \(a\) is not a specific type but a type variable which means that it can be substituted by any kind of types. If a type contains one or more type variables, the type is said to be \textit{polymorphic}, otherwise it is \textit{monomorphic}. In the examples given above, example 1 is a monomorphic type, while example 2 is a polymorphic type.

If we want a function to calculate the length of a number list, we can have a function type like following:

Example: \( \text{lenlist'} :: \quad [\text{num}] \rightarrow \text{num} \)

We can also get the same result by instantiating the type variable \(a\) in function \text{lenlist} with type \text{num}. We say \text{lenlist} is a more general function than function \text{lenlist'} or is polymorphic function. \text{lenlist'} can be said to be an instance function of \text{lenlist}.
Higher-Order Functions  One feature that makes a functional language a higher-
level language than other languages is the higher-order functions which are not available
in most imperative programming languages. Functional languages like Miranda are
higher-order languages which means that functions are first class citizens and can be
passed as parameters and returned as results.

*filter* is a built-in function in Miranda which takes a function as a parameter for the
first argument, the second argument has a type \([a]\) and the result type is \([a]\). The
definition of function *filter* is shown in Figure 1.

\[
\text{type variable}
\]
\[
filter :: (a \rightarrow \text{bool}) \rightarrow [a] \rightarrow [a]
\]
\[
\text{higher-order}
\]
\[
\begin{align*}
\text{filter } p \ [\ ] & = [] \\
\text{filter } p \ (\mathbf{x}:\mathbf{xs}) & = \mathbf{x} : \text{filter } p \ \mathbf{xs}, \quad \text{if } p \ \mathbf{x} \\
& = \text{filter } p \ \mathbf{xs}, \quad \text{otherwise}
\end{align*}
\]

Figure 1  An example of higher-order function

Recursion and Induction  Recursion is the continued repetition of the same oper-
ation or a group of operations (A recursive routine is one that uses itself as a subroutine).
Induction is a proof technique which can be used for proving properties of natural num-
bers. Recursion and induction are important properties of functions which are frequently
used in functional programming.

Recursion can be summarized as follows:

1. Define the base case(s).

2. Define the recursive case(s):

   (a) reduce the problem to simpler cases of the same problem,

   (b) write the code to solve the simpler cases,
(c) combine the results to give the required answer.

Often when describing a function, there are infinitely many cases to consider. In conventional imperative programming languages, this is solved using a loop, but in functional languages there are no explicit looping constructs. Instead, solutions to such problems are expressed by defining a recursive function (see examples in Figure 2 and Figure 3).

**Induction** can be described as follows:

1. the base case: a proof of \( P(0) \) or \( P(1) \);

2. the induction step: a proof of \( \forall n : \text{nat.} \ (P(n) \rightarrow P(n + 1)) \), in other words a general method that shows for all natural numbers \( n \) how, if you had a proof of \( P(n) \) (*induction hypothesis*), you could prove \( P(n+1) \).

To prove: \( \forall n : \text{nat.} \ P(n) \) (\( P \) holds for all natural numbers \( n \)).

Given these, you can indeed deduce \( \forall n : \text{nat.} \ P(n) \). This is the **Principle of Mathematical Induction**.

In functional programming language, induction can be applied to the following data types:

- Natural number
- Lists
- Structures

For the natural numbers, induction is simple. Here is a function \textbf{Sum} written in Miranda. \textbf{Sum} function sums up the natural number from \( 1 \) to \( n \). \textbf{Sum} is defined recursively and proved by induction (see Figure 2).
Introduction

\[ sum :: \text{num} \rightarrow \text{num} \]

\[
\begin{align*}
\text{sum 0} & = 0 & \text{base case} \\
\text{sum } n &= n + \text{sum } (n-1) \\
\end{align*}
\]

\[ \text{n-th case} \quad \text{(n-1) case} \]

\text{induction steps:}

\[
\begin{align*}
\text{base case} &: \ 1 \\
\forall n \ (\text{n-1 case}) &: \ \text{sum } (n-1) \\
\text{n-th case} &: \ \text{sum } n \\
\end{align*}
\]

Figure 2 Natural number induction

\textit{List} induction is induction applied to the length of a list, not each element in the list. In functional languages, quite often, the empty list is chosen as the base in the function definition for the function with list type as argument. An example for list induction is given in Figure 3.

Induction on structures is usually applied to recursive data types, like \textit{tree} type. The \textit{list} type is the basic structured type in functional languages.

\[ sum :: [a] \rightarrow \text{num} \]

\[
\begin{align*}
\text{lenlist } [] &= 0 & \text{base case} \\
\text{lenlist } (x:xs) &= 1 + \text{lenlist } xs \\
\end{align*}
\]

\[ \text{n-th case} \quad \text{(n-1) case} \]

\text{induction steps:}

\[
\begin{align*}
\text{base case} &: \ [ ] \\
\forall n(\text{n-1 case}) &: \ [x] \\
\text{n-th case} &: \ n \ case \ [x:xs] \\
\end{align*}
\]

Figure 3 Lists induction

1.3 An Overview of WISER

WISER is a reusable software base system developed by Deoraj Ramijisingh [15] in a functional programming environment. It is an interactive system that is directed at the first two stages of reuse-oriented program development. The concerns of WISER are
centered around the construction of functional programs (Miranda programs) and as such maintains an evolving structured software repository of primitive Miranda components. The main idea of system design is based on the functional generalization, specialization and freezing argument relationships. There are two kinds of relations for functional generalization, the first being general functions while the second is functions with more arguments. A function \( A1 \) is said to be more general than a function \( A2 \), if \( A2 \) is an instance of \( A1 \). The general function could perform the same operation as its instance function. Functional specialization is when a function \( F1 \) is said be more specific than a function \( F2 \), if \( F1 \) is an instance of \( F2 \). It means that the function \( F1 \) can be used to replace the function \( F2 \) by generalizing \( F1 \)'s source code. In a functional programming environment, functions with more arguments sometimes can behave the same or similar to the required functions by freezing the extra arguments. This is because of the feature of partial application in functional languages.

Based on these relationships among the functions, the software library was designed and developed to provide retrieval based on function types and associated management facilities. Components are classified based on the number of arguments, and stored in a record structure. The record structure is shown in Table 2.

\[
\sum \{F_\Sigma\} \{G_\Sigma\} \{S_\Sigma\} \{H_\Sigma\}
\]

- \( \sum \) represents the type
- \( F_\Sigma \) represent all the functions with type \( \Sigma \)
- \( G_\Sigma \) represents the offsets of all the immediate most generatral type(s) of \( \Sigma \)
- \( S_\Sigma \) represents the offsets of all the immediate most specific type(s) of \( \Sigma \)
- \( H_\Sigma \) represents the offsets of all the immediate extra-argument type (s) of \( \Sigma \)

Table 2 Record structure [15]

Retrieval plays an important role in software reuse. WISER provides a retrieval which uses function types as search keys. When doing a retrieval, the user needs to
provide a query which includes the number of arguments, the type information for each argument and the function result type.

\[ \text{Query: } num \rightarrow num \rightarrow num \]

\[ \text{arg1} \quad \text{arg2} \quad \text{Result type} \]

Figure 4 Query example 1.

\[ \text{Query: } [num] \rightarrow num \]

\[ \text{arg1} \quad \text{Result type} \]

Figure 5 Query example 2.

In Figure 4, the query means that functions have two input arguments with number type and the result with a number type. In Figure 5, the query means that functions have one input argument with list type and the result with number type. The retrieval results include all functions whose type exactly matches the user's query, more general functions whose type is compatible with the query, more specific functions, which means that the type of the function is more specific than the query type, and freezing arguments functions which means that these functions can have the same or similar behavior as required functions by freezing extra arguments.

Insertion and deletion provides the basic management mechanism for the library system. Insertion here is to insert a function type and associated links to this function so that the library system can be kept up to date. Deletion is to remove a function type and associated links to the function which is regarded unnecessary. In this way, the library can house good quality components for reuse.
1.4 Overview of the thesis

1.4.1 Motivation

Software reuse provides another way of software development with better reliability and lower cost and efforts. It offers a great deal of potential in terms of software productivity and software quality. However, reuse is only effective if it is easier to retrieve reusable components than to write them from scratch. So effective methods for retrieving components satisfying given requirements become increasingly important.

The work done in this thesis is to explore a practical method for retrieving executable components by executing them on the sample test data generated by the system and determining the components to be retrieved by verifying the results. Particularly, the work focuses on the method of test data generation and testing.

1.4.2 The Objectives of the Thesis Work

The objective of this thesis is to investigate the use of automated retrieval approach, Execution-based Retrieval and to improve the precision of retrieval in terms of the behavior of components. Testing and test data generation are the main parts of this thesis work. A prototype system was designed and implemented based on this idea in a functional programming environment. The system is designed to be easy to use with a GUI based user interface, associated management facilities, and to be able to interface with WISER and other retrieval systems. Finally this retrieval system together with the existing WISER system is linked to the World Wide Web through the Common Gateway Interface (CGI) under a client/server environment. People can access this library system and retrieve the reusable components by using a Web browser.

1.4.3 Thesis Statement

- Investigation of a practical method of retrieving reusable software components
based on execution.

☐ Implementation of a prototype system based on our method for the Miranda functions in a functional programming environment.

☐ Incorporating this retrieval system with the existing software base system (WISER) and linking it to the World Wide Web to make this library system public over Internet.

1.4.4 Organization of the Thesis

This thesis is organized into five chapters.

Chapter 1 gives a brief overview of background knowledge involved in the thesis work. That includes software reuse, the functional programming environment, an overview of software base system WISER and a brief overview of the thesis.

Chapter 2 explains the details of the execution-based retrieval approach, the process of retrieval and the test data generation. Run-time problems are also discussed.

Chapter 3 focuses on the design and implementation of this prototype system developed based on our approach which mainly involves the process of retrieval, test data generation, executing Miranda functions and verification. Some examples are given to show how the system works and how the retrieval is achieved based on our approach. Other features involving the implementation of this system are also addressed.

Chapter 4 focuses on the design and implementation of linking this prototype system to the World Wide Web. Basic concepts and features of Internet, protocols, and WWW are also introduced.

Chapter 5 gives the conclusions of this thesis and the future work in this area.

Finally there is a appendix of the Web pages for the WISER system.
Chapter 2
An Approach to Execution-based Retrieval

Location/Retrieval of reusable software components satisfying given requirements is a critical step for reuse after a reusable library is built up. An important property of software is that the software can be executed to transfer inputs into outputs. The main idea of our approach is that software components are identified by executing the components on sample test data generated by the system and retrieved by verifying the results.

This chapter addresses more details on the different phases involved in this approach, particularly on test data generation.

2.1 The Process of Execution Based Retrieval

The basic idea of our approach is really concerned with functional program testing process. In functional program testing, a program is considered to be a function in terms of input values and corresponding output values. It involves selecting elements from the program’s input domain, executing the program on the test cases, and comparing the actual outputs with the expected outputs.

The process of Execution-based Retrieval can be divided into four phases: The Selection of candidate components, Test data generation, Execution of components, and Verification of results. The whole process of this retrieval is shown in Figure 6.

![Diagram of the process of Execution-based Retrieval](image)

Figure 6 The process of Execution-based Retrieval
First the user is responsible for querying the system. Query here means to provide function types. Based on the query, the system will search the software library to find out the components whose type matches the query or whose type is compatible with the query to be the candidate functions. These components will be tested later. Second, the most important step is test data generation. After candidate functions are selected from a software library, the system will generate test cases based on the type of functions provided in the query and certain methods of test data generation which will be discussed in the following sections and interactively asks the user to provide the corresponding outputs on these test data. The third phase is execution. In this stage, the system will execute each candidate function on the test data to produce the system results. Finally, the system results and the results provided by the user are compared to determine the functions to be retrieved.

2.2 The Selection of Candidate Functions

The selection of candidate functions is done in the first phase in this retrieval method. The system is not going to execute each component in the software library. We need first to find which functions have the same properties on certain aspects. We call these functions candidate functions which will be tested later.

The type system adopted for a programming system has significant impact on the ways component can be combined. Functional language (like Miranda) is polymorphic strong typing system. It means that every expression or subexpression has a type, which can be deducted at compile time. By taking this advantage, candidate functions can be selected based on function types which means that the components selected from library have the same type or more general type as the query. Retrieval using function types as the search keys is best described and implemented in [15]. Selected functions are
type matched to user's query or type compatible with user's query. In another words, candidate functions include the exactly matched functions and more general functions [15]. An example is shown in Figure 7.

![Candidate functions table]

<table>
<thead>
<tr>
<th>Function Type</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>num→num→num</td>
<td>subtract</td>
</tr>
<tr>
<td>num→num→num</td>
<td>add</td>
</tr>
<tr>
<td>num→num→num</td>
<td>divid</td>
</tr>
<tr>
<td>num→num→num</td>
<td>power</td>
</tr>
<tr>
<td>num→num→num</td>
<td>gcd</td>
</tr>
<tr>
<td>num→num→num</td>
<td>times</td>
</tr>
<tr>
<td>num→num→num</td>
<td>perms</td>
</tr>
<tr>
<td>a→a→a</td>
<td>min2</td>
</tr>
<tr>
<td>a→a→a</td>
<td>max2</td>
</tr>
<tr>
<td>a→a→a</td>
<td>const</td>
</tr>
</tbody>
</table>

Figure 7 Selection of candidate components

2.3 The Strategy of Test Data Generation

Test data generation is a critical step in functional program testing. Functional program testing requires that the selection of test data can be made on the basis of the important properties of the elements in the domains of a program's input and output variables. One method of test data generation mentioned in sampling behavior [14] is that test data are generated randomly from the domain of argument. One problem is that random generation of test data may involve a large test data set which means that it may take a long time to test functions on these test data. Also, using random sampling needlessly reduces the precision of retrieval in some cases.

Test data generation here focuses on basic types and list type. In test data generation, we refer to the properties of recursion and induction being frequently used in functional programming languages and the principle of test data generation based on important properties of element domains of a program's input to simulate the function definition in functional programming.
The main idea in test data generation here is that first we fix a base case for each basic type, and then generate the second test case based on the type of argument. The third test case is generated based on the second test case.

2.3.1 Test Cases for Basic Types

\textit{num}, \textit{char}, \textit{bool} are well known primitive data types and list type is a basic structured type in functional languages. In this part of the thesis, we'll focus on the discussion of test data generation for these data types. Test data generation for \textit{tuple} type is also addressed.

\subsection*{2.3.1.1 Number Type}

The data type \textit{num} consists of whole numbers (or integers) and fractional numbers (also called floating-point number). A whole number is a number whose fractional part is zero.

Zero has a special meaning for number type. Quite often it is chosen to be the terminated case in a recursively defined function. So we chose zero as the base test case, and then selected a value \( n \) from the domain of argument randomly as the second test case. The third test case is generated by adding one to the second test case. Figure 8 shows the test cases for \textit{num} type and an example is shown in Figure 9.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure8.png}
\caption{Test cases for \textit{num} type}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure9.png}
\caption{An example of test cases for \textit{num} type}
\end{figure}

\textit{Query: }\textit{num} \rightarrow \textit{num}

Here \( m = 4 \)
2.3.1.2 Boolean Type

For boolean type bool, there are only two values True and False, so the test data is chosen either from True or False.

<table>
<thead>
<tr>
<th>Test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool type</td>
</tr>
<tr>
<td>The number of case</td>
</tr>
<tr>
<td>(1)</td>
</tr>
</tbody>
</table>

Figure 10 Test cases for bool type

2.3.1.3 Character Type

Test case for char type is chosen from ASCII set.

<table>
<thead>
<tr>
<th>Test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>char type</td>
</tr>
<tr>
<td>The number of case</td>
</tr>
<tr>
<td>(1)</td>
</tr>
</tbody>
</table>

Figure 11 Test cases for char type

2.3.1.4 List Type

By definition, a list is a linearly ordered collection of values. Like sequences in mathematics, a list can contain an infinite number of elements. For list type, all the elements of given list must be the same type. For example, a list can be a list of numbers, a list of characters, even a list of lists (of values all of the same type). Some examples are shown in Table 3.

<table>
<thead>
<tr>
<th>Value of a List</th>
<th>Type of the List</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 1, 2, 3 ]</td>
<td>[ num ]</td>
</tr>
<tr>
<td>[ False, False, True ]</td>
<td>[ bool ]</td>
</tr>
<tr>
<td>[ 'a', 'b', 'c' ]</td>
<td>[ char ]</td>
</tr>
<tr>
<td>[ [1,2], [], [9] ]</td>
<td>[ [num] ]</td>
</tr>
<tr>
<td>[ (1,2), (4,3), (0,10) ]</td>
<td>[ (num, num) ]</td>
</tr>
</tbody>
</table>

Table 3 Examples of list value
A list \([x]\) with just one element is known as a singleton list. Two lists are equal if and only if they have the same values with the same number of occurrences in the same order. Otherwise they are different. List type is a composite type constructed from other basic type or composite type. Some examples are shown in Table 4.

<table>
<thead>
<tr>
<th>List Type</th>
<th>Element Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>([ \text{num} ])</td>
<td>\text{num}</td>
</tr>
<tr>
<td>([ \text{bool} ])</td>
<td>\text{bool}</td>
</tr>
<tr>
<td>([ \text{char} ])</td>
<td>\text{char}</td>
</tr>
<tr>
<td>([ [\text{num}] ])</td>
<td>([\text{num}])</td>
</tr>
<tr>
<td>([ (\text{num, num}) ])</td>
<td>((\text{num, num})) \text{ tuple}</td>
</tr>
</tbody>
</table>

Table 4 Examples of list type

Test data generation for the list type can be divided into two steps: first we generate three cases for the length of the list. Zero is chosen as the base case for the length of the list. Second test case for the length of list is chosen from natural numbers and the third test case for the length of list is generated by adding one to the second case. Second, each element of the list is generated based on the type of element in the list. Test cases for list type are shown in Table 5.

<table>
<thead>
<tr>
<th>The type of list</th>
<th>The length of the list</th>
<th>The values of test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>([ \text{num} ])</td>
<td>0</td>
<td>([\ ])</td>
</tr>
<tr>
<td></td>
<td>(n)</td>
<td>([a_1, a_2, ..., a_n])</td>
</tr>
<tr>
<td></td>
<td>(n+1)</td>
<td>([b_1, b_2, ..., b_n, b_{n+1}])</td>
</tr>
<tr>
<td>([ \text{bool} ])</td>
<td>0</td>
<td>([\ ])</td>
</tr>
<tr>
<td></td>
<td>(n)</td>
<td>([c_1, c_2, ..., c_n])</td>
</tr>
<tr>
<td></td>
<td>(n+1)</td>
<td>([d_1, d_2, ..., d_n, d_{n+1}])</td>
</tr>
<tr>
<td>([ \text{char} ])</td>
<td>0</td>
<td>([\ ])</td>
</tr>
<tr>
<td></td>
<td>(n)</td>
<td>([e_1, e_2, ..., e_n])</td>
</tr>
<tr>
<td></td>
<td>(n+1)</td>
<td>([f_1, f_2, ..., f_n, f_{n+1}])</td>
</tr>
</tbody>
</table>

Table 5 Test cases for list type
Here a_i and b_i are the values of type \textit{num}, c_i and d_i are the values of type \textit{bool}, e_i and f_i are the values of type \textit{char}. An example of test data for number list is shown in Figure 12 and an example of test data generation for boolean list is shown in Figure 13.

\begin{center}
\begin{tabular}{|c|c|c|}
  \hline
  The number of case & Length of list & The value of test case \\
  \hline
  (1) & 0 & [1] \\
  (2) & 3 & [1.5, 7] \\
  (3) & 4 & [3, 5, 7, 2] \\
  \hline
\end{tabular}
\end{center}

\textbf{Query:} \texttt{[num] \rightarrow num} \hspace{2cm} \textbf{Test cases}

Here n = 3

\begin{center}
\begin{tabular}{|c|c|c|}
  \hline
  The number of case & Length of list & The value of test case \\
  \hline
  (1) & 0 & [1] \\
  (2) & 2 & [True, False] \\
  (3) & 3 & [False, False, True] \\
  \hline
\end{tabular}
\end{center}

\textbf{Query:} \texttt{[bool] \rightarrow num} \hspace{2cm} \textbf{Test cases}

Here n = 2

\begin{center}
\begin{tabular}{|c|c|c|}
  \hline
  The number of case & Length of list & The value of test case \\
  \hline
  (1) & 0 & [1] \\
  (2) & 3 & [1.5, 7] \\
  (3) & 4 & [3, 5, 7, 2] \\
  \hline
\end{tabular}
\end{center}

\textbf{Figure 12} An example of test cases for \textit{num} list

\textbf{Figure 13} An example of test cases for \textit{bool} list

\subsection{2.3.1.5 Tuple Type}

Tuple type is a pair of values. The type \texttt{(num,char)} consists of all pairs of values for which the first component is a number and the second is a character. In particular, (3, 'a') and (17.3, '+') are both values of type \texttt{(num, char)}. The type \texttt{\langle a, \beta \rangle} corresponds to the Cartesian product operation of set theory, where the notation \( \alpha \times \beta \) is more often seen.

Tuple type is quite similar to the list type in terms of that they both are composite types. Tuple type is structured from other basic types or composite types by pairing them together. The type of each component of pairs may different. Test data generation for tuple type can be done by generating test data for each component of the pairs and then pairing these values together. Test data generation for tuple type is treated as two arguments case. Test data generation for functions with more than one arguments (combination values) will be discussed in the next section. The following is an example
of test data generation for a tuple type. Test cases for each component of the tuple pair are shown in Table 6 and the final test data for the query given below are shown in Table 7.

Example: Query: \((\text{num}, \text{bool}) \rightarrow \text{num}\)

<table>
<thead>
<tr>
<th>Numbers of Test cases</th>
<th>Test case for first component of tuple</th>
<th>Test case for second component of tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>(2)</td>
<td>3</td>
<td>False</td>
</tr>
<tr>
<td>(3)</td>
<td>4</td>
<td>False</td>
</tr>
</tbody>
</table>

Table 6 Test cases for each component of a tuple pair \((\text{num}, \text{bool})\)

<table>
<thead>
<tr>
<th>Numbers of Test cases</th>
<th>Test case for tuple type ((\text{num}, \text{bool}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(0, False)</td>
</tr>
<tr>
<td>(2)</td>
<td>(3, False)</td>
</tr>
<tr>
<td>(3)</td>
<td>(4, False)</td>
</tr>
</tbody>
</table>

Table 7 The final test cases for the query \((\text{num}, \text{bool}) \rightarrow \text{num}\)

Now we have discussed the method of test data generation for basic data types, list type and tuple type. For more complex data types constructed from these basic data types, like a list of lists \([[a]]\), a list of tuples \((a,b)\) etc., test data generation can be done recursively based on these basic data types. An example is given in Table 8.

Example: Query: \([[\text{num}]] \rightarrow \text{num}\)

<table>
<thead>
<tr>
<th>Numbers of test cases</th>
<th>Test cases for argument ([[\text{num}]])</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>[ ]</td>
</tr>
<tr>
<td>(2)</td>
<td>[[1,2], [3,0]]</td>
</tr>
<tr>
<td>(3)</td>
<td>[[2,3,4], [3,5], [5,7,8]]</td>
</tr>
</tbody>
</table>

Table 8 An example of test data generation for more complex data type \([[\text{num}]] \rightarrow \text{num}\)
2.3.2 Test cases for Polymorphic Types

Sometimes the query provided by the user contains type variables which means that the user is looking for some general functions or polymorphic functions. In this case, test data generation can be done by first instantiating the type variables with some basic data types and then generating test cases based on the instantiated query. The following is an example.

Example 1: Query: \([a] \rightarrow [a]\)

For this query, we first instantiate the type variable \(a\) with the basic data type. Suppose the type variable \(a\) in this query is substituted by the type \(num\). The instantiated query becomes:

**Instantiated query:** \([num] \rightarrow [num]\)

The test data generated by the system are shown in Table 9.

<table>
<thead>
<tr>
<th>Numbers of test cases</th>
<th>The values of test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>[ ]</td>
</tr>
<tr>
<td>(2)</td>
<td>[3, 4, 6]</td>
</tr>
<tr>
<td>(3)</td>
<td>[9, 6, 5, 3]</td>
</tr>
</tbody>
</table>

Table 9 Test data generation for query \([a] \rightarrow [a]\)

Example 2: Query: \(a \rightarrow b \rightarrow a\)

For this query, the system first instantiates the query by substituting the type variables \(a\) with type \(num\) and \(b\) with type \(bool\). So the instantiated query becomes:

**Instantiated Query:** \(num \rightarrow bool \rightarrow num\)

Test data generated by the system for each argument are shown in Table 10.
2.3.3 Test cases for the Functions with more than one Arguments

When the user’s query contains more than one arguments, the combination values occur. In this case, test data generation can be done by generating test data for each argument first and then combining each case of each argument together. An example is shown in Table 11.

Example:  

<p>| Query: num --&gt; [num] --&gt; num |</p>
<table>
<thead>
<tr>
<th>Test cases for Argument 1</th>
<th>Test cases for Argument 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[ ]</td>
</tr>
<tr>
<td>n</td>
<td>[a₁,a₂, ... aₙ]</td>
</tr>
<tr>
<td>n+1</td>
<td>[b₁,b₂, ... bₙ, bₙ+1]</td>
</tr>
</tbody>
</table>

Table 11 Test cases for each argument of the function

In this query, the function takes two input arguments. The first argument has a type num. The second argument has a type [num] and the result in type num. Test data generated by the system for each argument are shown in Table 11. Here a₁ and b₁ are the values of type num. The final test cases for this query are shown in Table 12.
<table>
<thead>
<tr>
<th>Numbers</th>
<th>Argument 1</th>
<th>Argument 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>0</td>
<td>[ ]</td>
</tr>
<tr>
<td>(2)</td>
<td>0</td>
<td>[a_1, a_2, \ldots, a_n]</td>
</tr>
<tr>
<td>(3)</td>
<td>0</td>
<td>[b_1, b_2, \ldots, b_n, b_{n+1}]</td>
</tr>
<tr>
<td>(4)</td>
<td>n</td>
<td>[ ]</td>
</tr>
<tr>
<td>(5)</td>
<td>n</td>
<td>[a_1, a_2, \ldots, a_n]</td>
</tr>
<tr>
<td>(6)</td>
<td>n</td>
<td>[b_1, b_2, \ldots, b_n, b_{n+1}]</td>
</tr>
<tr>
<td>(7)</td>
<td>n+1</td>
<td>[ ]</td>
</tr>
<tr>
<td>(8)</td>
<td>n+1</td>
<td>[a_1, a_2, \ldots, a_n]</td>
</tr>
<tr>
<td>(9)</td>
<td>n+1</td>
<td>[b_1, b_2, \ldots, b_n, b_{n+1}]</td>
</tr>
</tbody>
</table>

Table 12 Test cases for the components with more than one arguments

For each basic type, there is a fix number of test data. The number of test cases for each basic type is shown in Table 13.

<table>
<thead>
<tr>
<th>Type</th>
<th>Values of test data</th>
<th>Number of test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>num</em></td>
<td>0, n, n+1</td>
<td>3</td>
</tr>
<tr>
<td><em>bool</em></td>
<td>True or False</td>
<td>1</td>
</tr>
<tr>
<td><em>char</em></td>
<td>Any characters</td>
<td>1</td>
</tr>
<tr>
<td>[<em>a]</em></td>
<td>[ ], [a_1, a_2, \ldots, a_n], [b_1, b_2, \ldots, b_n, b_{n+1}]</td>
<td>3</td>
</tr>
<tr>
<td>(<em>a</em>, <em>b</em>)</td>
<td>Depends on the type <em>a</em> and type <em>b</em></td>
<td><em>n_a x n_b</em></td>
</tr>
</tbody>
</table>

Table 13 Number of test cases for basic type

Here *a* and *b* are type variables and *n_a* and *n_b* are the number of test cases for type *a* and type *b* respectively.

### 2.3.4 Test Cases for Higher-Order Functions

Higher-order function plays an important role in functional languages. Higher-order functions are the functions which take a function as argument or return a function as the result. In this case, test case for the argument with function type should be functions.

Test data generation for higher-order functions can be done by first doing retrieval based
on the type of argument which has function type to get the functions whose type matches the argument type as test cases, and then generating test cases for the arguments which have non-function types.

**Example:**  
*Query:*  
\[(a \rightarrow \text{bool}) \rightarrow [a] \rightarrow [a]\]

This query asks the system to search for a higher-order function. The type of the first argument is a function with type \(a \rightarrow \text{bool}\) which takes one input argument and return a boolean type, the second argument is a list.

When the system handles this query, first the type variable \(a\) is instantiated, for instance, with type \(\text{num}\). So the instantiated query becomes:

**Instantiated Query:**  
\[(\text{num} \rightarrow \text{bool}) \rightarrow [\text{num}] \rightarrow [\text{num}]\]

<table>
<thead>
<tr>
<th>Test cases for Argument 1</th>
<th>Test cases for Argument 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>odd</td>
<td>[ ]</td>
</tr>
<tr>
<td>even</td>
<td>([a_1, a_2, \ldots a_n])</td>
</tr>
<tr>
<td>prime</td>
<td>([b_1, b_2, \ldots b_n, b_{n+1}])</td>
</tr>
</tbody>
</table>

*Table 14: Test cases for each argument*

The test cases for the first argument is generated by doing retrieval based on the function type \(\text{num} \rightarrow \text{bool}\). Functions in the library with type \(\text{num} \rightarrow \text{bool}\) and functions whose type is compatible with type \(\text{num} \rightarrow \text{bool}\) are selected to be the test cases for the first argument. Test cases generated by the system for each argument are shown in Table 14. The final test data can be formed by combining each test case of each argument together (see Table 15).
<table>
<thead>
<tr>
<th>Numbers of test cases</th>
<th>Argument 1</th>
<th>Argument 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>odd</td>
<td>[ ]</td>
</tr>
<tr>
<td>(2)</td>
<td>odd</td>
<td>[a₁, a₂, ..., aₙ ]</td>
</tr>
<tr>
<td>(3)</td>
<td>odd</td>
<td>[b₁, b₂, ..., bₙ,bₙ₊₁ ]</td>
</tr>
<tr>
<td>(4)</td>
<td>even</td>
<td>[ ]</td>
</tr>
<tr>
<td>(5)</td>
<td>even</td>
<td>[a₁, a₂, ..., aₙ ]</td>
</tr>
<tr>
<td>(6)</td>
<td>even</td>
<td>[b₁, b₂, ..., bₙ,bₙ₊₁ ]</td>
</tr>
<tr>
<td>(7)</td>
<td>prime</td>
<td>[ ]</td>
</tr>
<tr>
<td>(8)</td>
<td>prime</td>
<td>[a₁, a₂, ..., aₙ ]</td>
</tr>
<tr>
<td>(9)</td>
<td>prime</td>
<td>[b₁, b₂, ..., bₙ,bₙ₊₁ ]</td>
</tr>
</tbody>
</table>

Table 15 Test cases for query: (num --> bool) --> [num] --> [num]

Here, aᵢ and bᵢ are the values of type num. For example, if we want to retrieve the function filter, we can provide the corresponding results for each test case. Some sample data are given in Table 16 for retrieving function filter.

<table>
<thead>
<tr>
<th>Numbers of test cases</th>
<th>Argument 1</th>
<th>Argument 2</th>
<th>The values of results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>odd</td>
<td>[ j ]</td>
<td>[ ]</td>
</tr>
<tr>
<td>(2)</td>
<td>odd</td>
<td>[2, 3, 6 ]</td>
<td>[3]</td>
</tr>
<tr>
<td>(3)</td>
<td>odd</td>
<td>[3, 5, 2, 4]</td>
<td>[3, 5]</td>
</tr>
<tr>
<td>(4)</td>
<td>even</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
<tr>
<td>(5)</td>
<td>even</td>
<td>[2, 3, 6 ]</td>
<td>[2, 6]</td>
</tr>
<tr>
<td>(6)</td>
<td>even</td>
<td>[3, 5, 2, 4]</td>
<td>[2, 4]</td>
</tr>
<tr>
<td>(7)</td>
<td>prime</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
<tr>
<td>(8)</td>
<td>prime</td>
<td>[2, 3, 6 ]</td>
<td>[2, 3]</td>
</tr>
<tr>
<td>(9)</td>
<td>prime</td>
<td>[3, 5, 2, 4]</td>
<td>[3, 2, 5]</td>
</tr>
</tbody>
</table>

Table 16 Test cases and the corresponding results for retrieving function filter

2.4 Execution and Verification

Execution is the third phase of Execution-based Retrieval. The main task here is to execute each candidate function selected from the library on the each test case generated
by the system to produce the system results.

Since the retrieval is execution based, run-time problems are avoidable. Three problems arise immediately: error termination, non-termination and long run-time. The system handles these run-time problems from two aspects: Validation check and Time-out interval.

Validation check is mainly applied to input data to avoid invalid input leading run-time problems. This is because the system provides two ways for test data. One way is that test data come from user, which benefits to the user with good knowledge on library components. Another one is that the test data are generated by the system.

A good retrieval system should provide the user with results as soon as possible. Time-out interval is mainly used to solve long run-time problem and non-terminated case. If running a component takes a long time, the system will interrupt the execution before the process is completed and keep this component as candidate for next testing.

Verification is the last phase of this retrieval process. To determine the components to be retrieved, we compare the results produced by the system with the known results provided by the user on each test case. If the system results match the user’s results on each test case, then these functions are presented to the user. An example of verification of results is shown in Figure 14. In this example, the system compares each system result with the each known result provided by the user on each test case [16, 28] and [24,31,27]. Here we find that the function lenlist is the component required by the user.
Query:

\[ \text{[num]} \rightarrow \text{num} \]

<table>
<thead>
<tr>
<th>Candidate functions</th>
<th>testing</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{num}] \rightarrow \text{num})</td>
<td>sum</td>
<td></td>
</tr>
<tr>
<td>([\text{a}] \rightarrow \text{a})</td>
<td>last</td>
<td></td>
</tr>
<tr>
<td>([\text{a}] \rightarrow \text{a})</td>
<td>max</td>
<td></td>
</tr>
<tr>
<td>([\text{a}] \rightarrow \text{num})</td>
<td>lenlist</td>
<td></td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
<td></td>
</tr>
</tbody>
</table>

Comparing

<table>
<thead>
<tr>
<th>Test data</th>
<th>User's results</th>
<th>Candidate functions</th>
<th>System results</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\ldots])</td>
<td>0</td>
<td>sum, last, lenlist, max</td>
<td>0</td>
</tr>
<tr>
<td>([15,28])</td>
<td>2</td>
<td>sum, last, lenlist, max</td>
<td>44, 28</td>
</tr>
<tr>
<td>([24,31,27])</td>
<td>3</td>
<td>sum, last, lenlist, max</td>
<td>82, 27, 3, 31</td>
</tr>
</tbody>
</table>

Figure 14 An example of verification of results
Chapter 3
A Prototype System

Based on the idea we proposed in this thesis, a prototype system was designed and developed in a functional programming environment. This prototype system was designed to be able to incorporate with the existing system WISER. It is an interactive retrieval system that is directed at second stage of the process of software reuse.

The main concern of this system focuses on exploiting the executable properties and improving the precision of retrieval for executable components. Associated management facilities for supporting this retrieval method are also provided.

This chapter covers the design of this retrieval system, and explains how the retrieval is accomplished. The evolving features of the system during implementation are also addressed.

3.1 System Design

A number of features were considered during the system design. The implementation of this system focuses on the following design principles:

1. Retrieval system should be an easy-to-use system with GUI based user interface.

2. In order to satisfy different kind of users (professional and nonprofessional), this retrieval system provides two ways for producing test data. One is that test data can be provided by the user. This is mainly for the users with good knowledge on library components. The other is that the system generates test data automatically.

3. Since components are evaluated by executing them on test data, so ready-to-
execute components should be maintained by the system.
4. Since the system design relies on the concept of maintaining a growing software library, associated management facilities for supporting this design idea and retrieval should also be provided. This includes insertion of component's source code, deletion of components to replace it with more efficient ones, and a browser which allows users to browse the source of components among the library components.

5. Since this system is execution based, a run-time exception handling is a must. This includes handling three run-time problems: error termination, non-termination and long run-time.

6. This retrieval system should support repeatable retrieval which means that retrieval can be done several times and each time retrieval is performed based on the current results. In this way, the number of candidates can be further narrowed down until the smallest set of candidates.

7. Finally, this retrieval system together with the existing WISER system is made public by linking it to the World Wide Web with Common Gateway Interface (CGI) and HTML Form.

An overview of the system design is shown in Figure 15.

3.2 Implementation

C language, Functional language Miranda and UNIX system programming are involved in the implementation of this prototype system. All library components are written in Miranda script. Some network concepts and features are also involved in implementation of linking this reuse system to the World Wide Web. The implementation of this prototype system includes retrieval, insertion, deletion, and a browser. A graphical user interface written in X/Motif is provided by the system and a Web pages interface designed
for whole system (WISER) is also available to allow users to do retrieval through the World Wide Web. The major work in the implementation focuses on the retrieval part.

![Diagram of system design](image)

Figure 15 An overview of the system design

### 3.2.1 Retrieval

The implementation of retrieval follows the process of execution-based retrieval: *The selection of candidate functions, test data generation, execution and verification*, and also the system is designed to allow the users to do retrieval repeatedly on the current results, so as the number of candidates can be further narrowed down. In this part of the thesis, we focus on the implementation of each phase of retrieval. The design of retrieval is shown in Figure 16.

![Diagram of retrieval design](image)

Figure 16 The design of execution-based retrieval
3.2.1.1 The Selection of Candidate Components

The selection of candidate functions is achieved by doing retrieval based on function types. Type information should be provided by the user in the query. Retrieval based on function types as search keys have been done in [15]. The process of type-based retrieval is shown in Figure 17.

After doing retrieval based on types, four products are produced: exact match functions, more general functions, more specific functions and extra arguments functions. Exactly matched functions and more general matched functions are chosen as candidate functions for testing.

Start

User query

Need any changes

Yes

No

S/W Library

Call insertion

Type exists

Yes

No

Exact functions if exists

General functions if exists

Specific functions if exists

Freezing arg. functions if exists

Stop

Figure 17 Retrieval based on types [15]
3.2.1.2 Test Data Generation

Based on our design principles, the system provides two ways for test data generation. One is that test data come from user. In this case, user will be asked to provide test data through the window interface. Users are free to chose test data from the domain of argument. So the selection of test data really depends on the user’s knowledge. For some users, they may have good knowledge on the source of inputs that can distinguish similar candidates. This way provides these users with convenience and flexibility. Another one is that test data come from the system. It means that test data are generated by the system automatically. Here we will focus on the second method.

Test data generated by system means that test data used to test candidate functions are generated by the system automatically based on the method we described in the previous chapter. That includes the following steps:

1. Set up base test cases for each basic data type.
2. Generate the second test case based on the argument type for each argument.
3. The third case is generated based on the second test case and the type of argument.

Table 17 shows the base test cases for each basic data type. The process of test data generation is shown in Figure 19 and the algorithm of test data generation is given in Table 18.

<table>
<thead>
<tr>
<th>Type</th>
<th>Base case</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>0</td>
</tr>
<tr>
<td>bool</td>
<td>True or False</td>
</tr>
<tr>
<td>char</td>
<td>any character</td>
</tr>
<tr>
<td>[a]</td>
<td>[] (empty list)</td>
</tr>
</tbody>
</table>

Table 17 Base cases for each basic data type
Function test_data_generation (number_argument, type_arguments, test_data)
for i = 0 .. number_argument
    check type of argument [i]
    num type: generate number value
    char type: generate char value
    bool type: generate boolean value
    tuple type: get first component of tuple pair
        type_value(number_argument, argument, value, value_number)
        test_data_generation(1, type_argument, temp_data)
        get second component of tuple pair
        type_value(number_argument, value, value_number)
        test_data_generation(1, type_argument, temp_data)
        pair each test cases of each components together to form tuple
    list type:
        number list: generate number list
        char list: generate char list
        bool list: generate boolean list
        more complex type: test_data_generation(number_argument, type_argument, temp_data)
end for loop
end test_data_generation

Table 18 The algorithm of test data generation

Start

Type check = User query

Setup base case
for each basic
data type

Generate test case
for each argument

What type?

num bool char list tuple Higher-order

Generate Generate Generate Generate num list Generate bool list
num bool char value value value

Generate char list Generate tuple list

Generate tuple retrieval

No complete test data
generation for all arguments?

Yes

Form test data

Stop,

Figure 18 Test data generation.
Test data generation for primitive types \( \text{num}, \text{bool}, \text{char} \) simply chooses values based on the method we explained in the previous chapter. Some examples are given in Figure 19. For more complex data types, like a list of lists, tuples, a list of tuples etc., test data can be generated recursively based on these basic types. The algorithm of test data generation is shown in Table 18.

<table>
<thead>
<tr>
<th>Number of test cases</th>
<th>The values of test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>([1])</td>
</tr>
<tr>
<td>2</td>
<td>([2, 4, 3])</td>
</tr>
<tr>
<td>3</td>
<td>([5, 4, 7, 1])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of test cases</th>
<th>The values of test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>((0, \text{True}))</td>
</tr>
<tr>
<td>2</td>
<td>((5, \text{True}))</td>
</tr>
<tr>
<td>3</td>
<td>((9, \text{True}))</td>
</tr>
</tbody>
</table>

Figure 19 Examples of test case for basic data type and tuple type

Following is an example of test data generation for more complex type \( [[\text{num}]] \).

\[
\begin{align*}
F & : [[\text{num}]] \rightarrow \text{num} \\
\end{align*}
\]

\[
\begin{align*}
[[\text{num}]] & \\
\end{align*}
\]

\[
\begin{align*}
[[\text{num}]] & \\
\end{align*}
\]

\[
\begin{align*}
\text{General list values of type } \text{num} & \\
\end{align*}
\]

\[
\begin{align*}
\text{General list values of type } \text{num} & \\
\end{align*}
\]

Figure 20 Data flow of test data generation for first argument of query \( [[\text{num}]] \rightarrow \text{num} \)

In this example, the argument type is \( [[\text{num}]] \), a list of lists of number type. First we check if argument type is list. If it is, then we further check the element type inside list. If the element type is still a composite type, like list, or tuple, we keep going until the basic data type is met. Based on the basic data type, test data are generated and then go up one level to generate the value of the list until to the top level. This is a recursive procedure (see Figure 20). Test data generated by the system are shown in Table 21.
When functions have more than one arguments, combination values occur. In this case, test data can be generated by first generating test cases for each argument and then combining these values together to form the final test cases. Following is an example of test data generation for the functions with more than one arguments.

**Example:**  
**Query:**  
\[ ([\text{num}]) \rightarrow \text{num} \rightarrow \text{num} \]

In this query, functions take two input arguments. The first argument has a type \([\text{[num]}]\). The second argument has a type \(\text{num}\) and the result type is \(\text{num}\). The test data generated by the system for each argument are shown in Table 19 and the final test data are shown in Table 20.

**Table 19** Test cases of each argument for the query \([\text{[num]}] \rightarrow \text{num} \rightarrow \text{num}\)
<table>
<thead>
<tr>
<th>Numbers of Test cases</th>
<th>The values for argument 1</th>
<th>The values for argument 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>[ ]</td>
<td>0</td>
</tr>
<tr>
<td>(2)</td>
<td>[[1, 2], [5, 3]]</td>
<td>0</td>
</tr>
<tr>
<td>(3)</td>
<td>[[4, 7, 9], [2,5,4], [3,2]]</td>
<td>0</td>
</tr>
<tr>
<td>(4)</td>
<td>[ ]</td>
<td>4</td>
</tr>
<tr>
<td>(5)</td>
<td>[[1, 2], [5, 3]]</td>
<td>4</td>
</tr>
<tr>
<td>(6)</td>
<td>[[4, 7, 9], [2,5,4], [3,2]]</td>
<td>4</td>
</tr>
<tr>
<td>(7)</td>
<td>[ ]</td>
<td>5</td>
</tr>
<tr>
<td>(8)</td>
<td>[[1, 2], [5, 3]]</td>
<td>5</td>
</tr>
<tr>
<td>(9)</td>
<td>[[4, 7, 9], [2,5,4], [3,2]]</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 20: Test data for the query [[num]] → num → num

3.2.1.3 Execution and Run-time Exception Handling

In this phase, the system will execute each candidate function to produce the system results. During the retrieval, system needs to frequently access Miranda system to run Miranda functions on test data. Since Miranda is running as a self-contained subsystem under UNIX, some UNIX system programming techniques are involved in communication between C language and Miranda system. When executing a function on test data, function name and test data need to be passed to Miranda system and after completing the execution, the result has to be sent back. That is done by using UNIX pipes in implementation.

Since the behavior of components is evaluated by executing them on test cases, the system should be able to handle run-time problems. The reasons which lead to run-time problems are various. In implementation, we provide exception handling from two aspects given below:

- Exception handling on input test data (Validation on input).
- Exception handling on output (Time-out interval).
Validation Check. Since the system allows the users to provide test data through window interface, illegal input may occur by typing mistakes or other mistakes. The purpose of validation on test data is to check if the inputs data for each argument matches its type which is provided in the query. Figure 22 shows the algorithm of validation check and an example is given in Figure 23.

```
start

test data \rightarrow Validation check \rightarrow argument type

\rightarrow Yes \rightarrow stop

\rightarrow No \rightarrow error message

\rightarrow type match?
```

Figure 22 Validation check

Query: \( \text{num} \rightarrow \text{num} \rightarrow \text{num} \)

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Type</th>
<th>Test Data</th>
<th>Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg 1</td>
<td>num</td>
<td>35</td>
<td>valid</td>
</tr>
<tr>
<td>arg 2</td>
<td>num</td>
<td>cs</td>
<td>not valid</td>
</tr>
</tbody>
</table>

Figure 23 Example 1 Validation on test data

To some degree, validation check plays a role to prevent some problems happen at run-time caused by illegal inputs.

Time-out Interval. A good retrieval system should present retrieval results to the users as soon as possible. In execution phase, sometimes it may take a long time to run a function on some test cases or some functions may not terminate. To handle this problem, we chose time-out interval technique to interrupt an execution of a function.
This can be done by using UNIX system programming techniques. The system will check the execution condition periodically during the execution stage. If the time of running a function exceed the certain period time setup by the system, this process will be terminated and the control will be back to the retrieval system.

### 3.2.1.4 Verification of Results

The last stage of retrieval is the verification. After all candidate functions are evaluated on test data, the system results are produced. The functions to be retrieved can be determined by comparing the system results with known results provided by the user on each test case. If the system results match the user's results on each test case, these functions are presented to the user. An example is shown in Figure 24.

![Diagram](image)

**Figure 24** An example of verification of results

---

### 3.2.2 Insertion

Under the purpose of maintaining a growing library system, the feature to insert the source codes of new components into the library should be provided by the system. The insertion here includes the function type and the source codes. Type insertion has been described and implemented in [15]. To support execution-based retrieval, the source
codes of new components should also be inserted. Figure 25 shows the algorithm for insertion of the source codes.

![Algorithm Diagram]

**Figure 25** Insertion

### 3.2.3 Deletion

As certain software components become obsolete or as errors are detected, some software components may no longer be good or useful. The reuse system must allow components to be removed from the software components storage area so that library can be maintained up-to-date. That includes removing the links to the component and the source codes. The process of deleting a component from the library is shown in Figure 26.

### 3.2.4 Saving and Browsing

To support reuse-based software development and help the user to get the required components for their own use, the system provides the facilities of saving and browsing the source codes of components. Saving provides users with capability of saving source
codes and browsing allows users to browse the library components based on the number of arguments.

![Flowchart](image)

3.3 User Interface and Examples

In this section, we describe the graphical user interface developed for this prototype system along with actual examples doing retrieval with this system.

The main concern during the design of interface is to try to provide users with an attractive, convenient and easy-to-use user interface for retrieval. All the interaction with the system is done through a window based I/O. This graphical user interface was written in X/Motif. The main window of this user interface for execution-based retrieval is interfaced with the existing system WISER. The organization of the window based interface is shown in Figure 27.
This user interface was designed so as to allow uniformity among windows and associated operations of library management can be also done through the window interface. The system provides on-line help at different level in this window hierarchy, so that the user can get help at any time through the on-line help menus. Three kinds of retrieval methods provided by this incorporated system are available to allow user to do retrieval by navigating through menus buttons. **Type-retrieve** performs the retrieval based on function types. **Sep-retrieve** performs the retrieval based on semantic properties and **Exe-retrieve** performs the retrieval based on the execution.

When the user invokes this system, the main window is pop-up (see Figure 28). The user can start interaction with system by using the mouse button-one within the main window. Seven buttons are designed in the menu bar in main window: **File, Retrieve, Insert, Delete, Browse, Version, Help.**
Under the File menu button, three pull down buttons, save, empty, and quit are designed to allow the user to save source, empty the text area in main window and quit the system. With Reactive menu button, three retrieval methods are identified by button Type-retrieve for type-based retrieval, Sep-retrieve for semantic-properties-based
retrieval and *Exe-retrieve* for execution-based retrieval. When the user clicks on these buttons, an associated retrieval session is invoked. With *Insert* and *Delete* buttons, the library system can be updated with this window-based interface.

Following discussion will focus on the execution-based retrieval part with actual examples.

### 3.3.1 Examples of Retrieval

We give two examples for doing retrieval with this system to show the results using our approach and how the system works to achieve retrieval.

#### 3.3.1.1 Retrieving function *sort*

Suppose that we want to retrieve the function *sort* which sorts a list. We can start retrieval by clicking on menu button *Retrieve* in main window. Three pull down buttons show up labeled by *Type-retrieval*, *Sep-retrieval*, and *Exe-retrieve*. Execution-based retrieval is invoked by pressing on button *Exe-retrieve*. Once the user presses this button, Execution-based retrieval window is pop-up to allow the user to enter the query.

In order to retrieve the function *sort*, the query we need to provide to the system can be like following:

\[
\text{Query: } [\text{num}] \rightarrow [\text{num}]
\]

This query means that the function takes one argument with number list and returns a number list. Entering the type for each argument and function result type can be done through Retrieve Window (see Figure29 ).
After the user submits this query to the system by clicking on button *Apply*, a number of candidate functions are produced by the system by doing type-based retrieval (see Figure 30). These candidate functions are type matched to the query or type compatible with the query. We can see that eight candidates are listed in the Candidate Window. From this window, the user can make the choice of doing user-test or system-test. User-test means that the system executes each candidate function on the test data provided by the user. System-test means that test data used in testing are generated by the system automatically. Let's do system-test first to see the result. System-test can be started by clicking on button *S-Test* in Candidates Window.
By using system-test dialog, the user is asked to enter the result values for each test case generated by the system. The user cannot see all the test data generated by the system at one time. But once the user provides the result value for the first test case which is shown in the list box and hits the Return key, the next test case is shown up in the list box. The test data are shown in the list box one by one each time when the user hits the Return key. The test data generated by the system and the corresponding result values provided by the user for this query are shown in Table 21 and Figure 31. The number of test cases varies depending on the query and the type of arguments. Before the user finishes supplying all the values of results for each test case, the system will not allow the user to start system test by locking the button Apply in the system-test window. After all required number of results are provided, the Apply button is released and testing can be started by pressing on it. We get sort function as the result which is shown in Match Function Window (see Figure 32).
We can see that the only function `sort` is listed in the Match Functions Window. The user can view the source codes of this function by clicking on the function name. The source of the function are shown in the text area in main window. Saving the source codes can be done by pressing button `Save` under the menu button `File` in main window. A small window is pop-up for entering the file name and the directory name which the user wants the source codes to save into (see Figure 33).

The Execution-based-Retrieval system supports repeatable retrieval which means that when the set of retrieval results is large, the user can do execution-based retrieval again.
over the current retrieval results to reduce the number of candidates. This procedure can be repeated several times to get the smallest set of candidates.

To facilitate this feature, two buttons *U-Test* and *S-Test* are designed in Match Functions window. The user can either go to user-test or system-test for doing retrieval again over the current results. *U-Test* goes to user-test, while, *S-Test* goes to system-test. The following retrieval can then be done by either doing the user-test or system-test depending on the user’s choice.

![Match Functions Window](image1)

**Figure 32** Match Functions Window

![Saving Window](image2)

**Figure 33** Saving Window
We have done the retrieval with system-test. Now let's see how the retrieval is accomplished by user-test. We use the same query to do the retrieval.

![User Test Window](image)

Figure 34 User Test Window

One flexibility provided by the execution-based retrieval system is that the user can choose test data for testing. This is specially for the users with good knowledge on library components. User-test can be invoked by pressing button *U-Test* in Candidates Window. We give the test data [4,3,5,2] and the value of result on this test data is [2,3,4,5] in user testing window (see Figure 34). Retrieval can be started by pressing button *Apply* in User-Test Window. System will test each candidate functions on the test data [4,3,5,2] and the retrieval results are shown in Match Function Window (see Figure 32).

3.3.1.2 Retrieving function *filter*

*filter* is a higher-order function. To retrieve function *filter*, we need to provide a query as follows:
Query: \((a \rightarrow \mathit{bool}) \rightarrow [a] \rightarrow [a]\) (see Figure 35)

Since the query contains type variable \(a\), so the system first instantiates the type variable \(a\), for instance, with type \(\mathit{num}\). The instantiated query becomes:

\((\mathit{num} \rightarrow \mathit{bool}) \rightarrow [\mathit{num}] \rightarrow [\mathit{num}]\)

The candidate functions are listed in Candidate Window (see Figure 36). We go to system-test and the values of test data generated by the system and the values of corresponding results provided by the user are shown in Table 22 and System-test window (see Figure 37). The result is listed in Match Functions Window (Figure 38).

---

Figure 35 Retrieval Window for query \((a \rightarrow \mathit{bool}) \rightarrow [a] \rightarrow [a]\)
<table>
<thead>
<tr>
<th>Numbers of test cases</th>
<th>Argument 1</th>
<th>Argument 2</th>
<th>The values of results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>odd</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
<tr>
<td>(2)</td>
<td>odd</td>
<td>[41, 38, 45]</td>
<td>[41, 45]</td>
</tr>
<tr>
<td>(3)</td>
<td>odd</td>
<td>[3, 5, 2, 4]</td>
<td>[3, 5]</td>
</tr>
<tr>
<td>(4)</td>
<td>even</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
<tr>
<td>(5)</td>
<td>even</td>
<td>[41, 38, 45]</td>
<td>[38]</td>
</tr>
<tr>
<td>(6)</td>
<td>even</td>
<td>[3, 5, 2, 4]</td>
<td>[2, 4]</td>
</tr>
<tr>
<td>(7)</td>
<td>prime</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
<tr>
<td>(8)</td>
<td>prime</td>
<td>[41, 38, 45]</td>
<td>[41]</td>
</tr>
<tr>
<td>(9)</td>
<td>prime</td>
<td>[3, 5, 2, 4]</td>
<td>[3, 2, 5]</td>
</tr>
</tbody>
</table>

Table 22 Test cases and the corresponding results for retrieving function `filter`.

Figure 36 Candidates Window for query `(a -> bool) -> [a] -> [a]`
3.3.2 Insertion and Deletion

In order to maintain a growing library system, insertion and deletion features are a must. Here insertion and deletion focus on inserting and deleting the source codes of
components. When the user wants to insert a new component, the type of the component, the semantic properties of the component and the source of the component may be all that needs to be inserted. So we use one window for doing all insertion. A button Source Code is designed in insertion window for inserting the source codes.

Suppose we want to insert a function $\text{exesqr}$ which calculates the square of a number. This function takes one input argument with type $\text{num}$ and the result with type $\text{num}$. We enter the type information through the Insert Window (see Figure 39). The source codes of the function are entered into the Source Window (see Figure 41) by clicking on button Source Code in Insert Window (see Figure 39).

![INSERT_popup](image)

**Figure 39** Insertion Window
Deletion allows the user to remove a component from the library. When the user clicks on button **Delete** in main window, deletion window is pop-up. The user can enter the function name and type what he/she wants to delete from the library. Once the button **Apply** is clicked, all information about this function is removed from the library. Deleting the function **exesqr** is shown in Figure 40.

![Figure 40 Deletion Window](image)

### 3.3.3 Browser

A browser allows the users to browse the source codes of library components based on the number of arguments. All functions with the number of arguments which is specified by the user are listed in the list box in Ex-Browser Window. To view the source codes of a function, just click on the function name, and the source codes are shown in the lower part of Ex-Browser Window. The browser window is shown in Figure 42.
Figure 41 Source Code Window

Figure 42 Browser Window
4.1 Internet and the World Wide Web

Internet is thought the biggest network in the world now. It offers public access to the network community. Finding information on the Internet is based on the principle of client/server architecture. The server is the computer system that contains information, such as electronic mail, database information, or text files. The client system requests the appropriate data from the server system. While a client system can travel anywhere in the world, it must ultimately attach to the server, whether to transfer files or to logon to another computer system. TCP/IP protocol is the standard that allows the client and server to communicate. Figure 43 shows a typical client/server model for data communication.

![Diagram of client/server model for data communication]

As the popularity of Internet increased, people become more aware of its potential. The World Wide Web (WWW) is a product of the continuous search for innovative ways of sharing resources through the Internet [5]. It was originally developed to allow information sharing within internationally distributed teams, and the dissemination of information by support groups. Like other Internet applications, the WWW is based on a client/server architecture. The client is called a browser and can be a PC, Macintosh,
or UNIX platform. Each WWW page is linked to other WWW pages with hyperlinks, words, phrases, or graphics that are underlined, highlighted, or otherwise indicated. Each hyperlink is linked to the WWW address of a page containing additional information on the particular subject. To retrieve and display documents, just point and click on a mouse button to one of thousands of WWW multimedia server pages located worldwide. Figure 44 shows the relationship between WWW and Internet.

![Diagram](image)

Figure 44 The Web within the Internet

World Wide Web is a universe of network-accessible information with body of software, and a set of protocols and conventions. The following features make WWW become currently the most advanced information system deployed on the Internet.

- Hypermedia Documents
- Hypertext Transfer Protocol (HTTP)
- Uniform Resource Locator
- Browsers

**Universal Readership and Hypermedia Documents.** WWW hypermedia documents are created using **Hypertext Markup Language (HTML)**, which is similar to the **Standard**
Generalized Markup Language (SGML), a standard publication language. Like SGML, HTML contains special format codes for document presentation. HTML documents support text graphics, audio, and video.

The principle of universal readership is that once information is available, it should be accessible from any type of computer, in any place. In practice, the WWW uses the concept of hypertext. Hypertext is text with links invented in 1965. It means that text is not constrained to be linear. MIME (Multipurpose Internet Mail Extensions) is to allow for multimedia components to be sent with electronic mail messages by specifying their types in advance so that the mail program could display them as part of the message. Client/Server model allows the Web to scale on the Internet. All the clients and all the servers are connected to each other by the Internet. A browser is the client in the Web’s Client/Server model. For a user, a browser is the gateway to the Web. A typical hypertext documents is shown in Figure 45.

![Hypertext documents](image)

Figure 45 Hypertext documents

Hypertext Transfer Protocol (HTTP). Hypertext Transmission Protocol (HTTP) is the standard language that WWW clients and servers use for communication. Sometimes WWW servers are referred to as HTTP servers. HTTP is also used as a path-name to
access a certain location. It’s very easy to locate the source of information on the WWW since every organization has its own location identifier or Uniform Resource Locator (URL). HTTP is a simple data transfer protocol that binds the Webs together based on a Client/Server model.

**Uniform Resource Locators (URL).** Uniform Resource Locators are the addresses of Web resources. The URL works in the following manner: The whole network is held together by links. The browser scampers over the TCP/IP with a TCP request to a URL. At the other end of the URL sits, an HTTP server that send the requested information back using HTTP. A simple URL address is shown in Figure 46.

![URL Diagram](http://www.cs.uwindsor.ca/cgi-bin/wiser.html)

**Figure 46** A URL diagram

### 4.2 Fill-out Form and Common Gateway Interface (CGI)

**Fill-out Form** is a HTML (*HyperText Makeup Language*) feature used in HTML documents to allow the user to enter information directly on a Web page, and then have it transmitted to a server. The server will process the information and forward it the page owner or to some other mechanism. HTML Forms create interactive areas in the Web pages. The data entered into a form by a user is assembled by the browser into a series of name/value pairs. These name/values pairs are sent to the server to be processed by a CGI script which is specified by the ACTION attribute in the form.

**Common Gateway Interface (CGI)** is a standard for interfacing external applications with information servers, such as HTTP or Web servers. Gateways are programs or scripts which handle information requests and return the appropriate document or even
generate a document on the fly. With CGI programs and HTML Forms, a database or any other information system can be made available world wide. A typical http client/server interaction with CGI is shown Figure 47.

Typical http Client-Server interaction

![Diagram of Client-Server interaction]

Figure 47 A schematic overview of data flow using CGI.

Any programming languages that can access UNIX environment variables can use the CGI. CGI programs can be written in following languages and scripts.

- C/C++
- Fortran
- PERL
- TCL
- Any Unix shell script
- Visual Basic
- AppleScript

### 4.3 Running Retrieval System Through CGI

Based on the concepts of CGI and HTML Forms, an application can be easily made available world wide. The user can send a request through a Web browser to access a application system or database and the application or database management system runs under CGI as a server to process the user’s request and dynamically present the results to the user. With CGI, and HTML Forms, each Web page can be generated dynamically.
Execution-based retrieval done in this thesis work together with the existing WISER system is linked to the World Wide Web with CGI programs. We call this Web system WISER.

The design of this Web system for WISER focuses on retrieval and browsing together with providing an introduction page about-WISER. For the security reason, this Web system does not provide direct insertion feature to the user. All insertions have to be done by the system administrator. But this Web system provides a web page to allow the users to post their component which they want to insert into our database, and the insertion can be done by the system administrator sometime later. The organization of this Web system designed for WISER is shown in Figure 48.

![Diagram of WISER Web pages organization](image)

**Figure 48** Organization of Web pages for WISER

In this Web page hierarchy, each page is generated dynamically with CGI programs except the WISER home page and the about-WISER page which are written in HTML. WISER home page provides the basic description of WISER and how to invoke this system. About-WISER page is an introduction to WISER system, which includes the
idea, design, features and implementation of this system. All other retrieval pages are generated dynamically based on the request from the user or the results produced by the retrieval system.

All parts of this library system along with CGI programs reside in a special directory called *cgi-bin*. When the user sends their request to the server, the server will pass user's request to CGI programs to process. CGI programs are executed on the server to produce the results base on user's requests and dynamically generate Web pages for presenting the results.

In WISER home page (see Figure 49), we provide basic information on reusable system. More details about the system can be found in about-WISER page which is invoked by clicking on the image title or the underlined hyperlink WISER. Three radio buttons *Browsing*, *Retrieval*, and *Insertion* are designed to allow users to chose what they want to do with this system. *Browsing* provides a feature to allow users to browse the library components based on the number of arguments. *Retrieval* allows users to retrieve the library components with different retrieval methods provided by the system. Finally the *Insertion* allows users to post their components to us to insert into our database. *Submit Query* button let user to send their request to the server. Three retrieval methods *Type-based Retrieval*, *Semantics-based Retrieval* and *Execution-based Retrieval* are designed and developed by this incorporated system. To do retrieval, the user need to chose one of retrieval methods and provide the query which includes the number of arguments, the type for each argument and the function result type. Then retrieval can be started by clicking on button *Submit Query*.

### 4.4 An Example of Retrieval Using the Web Browser

Here we give a simple example of using our Web interface to do retrieval. We use
the same query which is described in Chapter 4 to retrieve function `sort`.

**Example:**

\[
\text{Query: } [\text{num}] \rightarrow [\text{num}]
\]

First we need to get in Netscape and open location at the following address to invoke WISER Home Page.

http://www.cs.uwindsor.ca/ecta-meta-index/people/csgsds/bing/wiser.html

Clicking on radio button *Retrieval* to tell the system to do retrieval, select the number of arguments and then press button *Submit Query* to invoke the main retrieval page (see Figure 49). The main retrieval page shows up to allow the user to enter the type information for each argument and function return type (see Figure 50). By clicking on radio button *Execution Based Retrieval* and *Submit Query* to start execution-based retrieval. All candidate functions are listed in Execution Based Retrieval Page (see Figure 52). For each function, there is a link to allow the user to view the source by clicking on the function name. To do the retrieval with user-test by clicking on radio button *User Test* and system-test with radio button *System Test*. In System Testing Page shown in Figure 51, all the test data generated by the system are shown at left side of this page. The user needs to provide the values of results for each test case and press button *Submit Query* to start testing. In User Testing Page shown in Figure 53, the user needs to provide the test data and result value to the system and then clicks on button *Submit Query* to invoke the retrieval. The retrieval result is shown in Execution Based Retrieval Results Page (see Figure 54).
A Reusable Software Components Library

**WISER** is a reusable software components prototype system developed in functional programming environment (Miranda). There are about 150 components written in Miranda script stored in this library. You can retrieve components by using different retrieval methods provided by this library management system, **type based retrieval**, **semantics properties based retrieval** and **execution based retrieval**. Also this system provide a browsing and insertion features to allow user to browse library components based on the number of arguments and insert new components into database.

To use this system, you need to click on the radio button to decide what you want to do, then select the number of arguments for the components you want to retrieve, browse or insert and press "Submit Query" button to submit your query.

Please select the the number of arguments:

- Browsing
- Retrieval
- Insertion

Submit Query

ping@cs.uwindsor.ca
Linking the Retrieval System to the World Wide Web

Figure 50 WISER Retrieval Page

Figure 51 WISER System Testing page
Figure 52 WISER Execution Based Retrieval Page
Linking the Retrieval System to the World Wide Web

Figure 53 WISER User Testing Page

Figure 54 WISER Execution Based Retrieval Results Page
5.1 Conclusions

In this thesis, we have investigated the feasibility of retrieving software components by executing them on some test data generated by the system. In particular, we focused on the method of test data generation. This method works well for monotonic functions. We have three test cases $0, n, n+1$ in general. Other alternatives may also apply to this method, like $0, n, n+r$. Here $r$ is some other value than 1. Based on our approach, a prototype system was designed and developed in a functional programming environment. This retrieval system relied on the concepts of functional program testing to achieve retrieval. Various tools were also provided by this retrieval system to allow users to insert, delete, and browse the library source. Execution-based retrieval can be divided into four phases: *The selection of candidate functions, test data generation, execution and verification.* In chapter 2, we emphasized on the test data generation. In test data generation, we referred to the properties of recursion and induction being frequently used in functional languages and the principle of test data generation based on the important properties of element domains of to simulate function definition. Test data generation here focused on the basic data types and list type which are commonly used in programming.

Comparing our method of test data generation with random generation, one benefit of our method is that the test data set is quite small which means that it takes less time to run all candidate functions on these test data. Some special values are chosen as test cases (base case) like zero, the empty list which may better identify the behavior of components than random test in some aspects. Due to the execution-basis retrieval,
run-time exception handling was also discussed in this thesis. We provide two solutions for handling run-time problems: validation check and time-out interval.

This retrieval system was designed to support repeatable retrieval which means that retrieval could be done several times. Each time retrieval could be performed based on the current results, so that the number of candidates could be further narrowed down.

A graphical user interface written in X/Motif was designed and developed for this retrieval system. All operations of retrieval and library management could be done through the window based I/O. Also an on-line help menu was provided on each level in window hierarchy to help users in getting familiar with this retrieval system.

Finally this system together with the existing WISER system was linked to the World Wide Web under a client/server environment. Everybody can access this reuse system through a Web browser. For the security reason, the Web system designed for WISER focused on the retrieval and browsing the library components. With CGI and HTML Forms, the retrieval results could be dynamically presented to the user through the Web browser.

Based on the work carried out in this thesis, a number of conclusions can be formed:

1. This retrieval system provides a useful tool for program development based on reusable software components.

2. To some degree, execution-based retrieval proposed here improves the precision of retrieval in terms of the behavior of components.

3. This retrieval system support repeatable retrieval which means that retrieval can be done several times. Each time the retrieval can be performed based on the current results, so the number of candidates can be further narrowed down.
4. The system allows the user to update library components by providing insertion and deletion features.

5. This system is opened to the public by putting it on the Web with CGI programs and HTML Forms. So people can share the resources over the Internet.

5.2 Future Work

Execution-based retrieval of software components from a software library exploits the executable property of software. It does improve the precision of retrieval in terms of the behavior of components. However, when the number of candidate functions is large, it may take a long time to run all the candidate functions on test data. Therefore, this approach should be improved in some way so as to make the retrieval more efficient.

One way to improve the efficiency is to switch the execution part in retrieval to the insertion part. It means that execution is carried out during insertion and certain information are generated based on the results of execution. These information are maintained by the system for retrieval. Now retrieval can be done by matching information provided by the user to the information maintained by the system instead of executing components each time. Retrieval is heavily relied in reuse system, but insertion is not. In this way, the time needed to execute the components can be reduced. Also the test data generation for user defined data types may need some work to improve this retrieval method.

Another area of future work is retrieval based on functional composition. Currently the user can only retrieve the components which exist in the library. If no components satisfy the user’s requirements, the system cannot present any results to the user. In some cases, the components required by the user can be constructed from the existing library components. In this way, the reusability of software components can be improved.
BIBLIOGRAPHY


Appendix A  Wiser Web Pages

A.1 Wiser Home Page

Netscape: Wiser: A Reusable Software Library System

A Reusable Software Components Library

Wiser is a reusable software components prototype system developed in functional programming environment (Miranda). There are about 150 components written in Miranda script stored in this library. You can retrieve components by using different retrieval methods provided by this library management system: type based retrieval, semantic properties based retrieval and execution based retrieval. Also this system provides a browsing and insertion feature to allow user to browse library components based on the number of arguments and insert new components into database.

To use this system, you need to click on the radio button to decide what you want to do, then select the number of arguments for the component you want to retrieve, browse or insert and press "Submit Query" button to submit your query.

Please select the the number of arguments:

diamond Browsing  diamond Retrieval  diamond Insertion  1  
Submit Query

king@cs.uwindsor.ca

Figure 55 Wiser Home Page
### A.2 About-WISER Page

**About WISER**

WISER is a re usable software components prototype system developed in functional programming environment (Miranda). This system was originally designed and implemented in C language and Windows programming environment by Devaraj Ramalingam, Shailesh Khairil and Ying Bai. Graduate Students of School of Computer Science, at University of Windsor as part of our thesis work under the direction of our supervisor Dr. Y. Park. The implementation of linking this system to World Wide Web through CGI (Common Gateway Interface) was done by Ying Bai.

There are about 150 components stored in this library. You can retrieve components by using different retrieval methods provided by this library management system. **Type based retrieval**, **semantics based retrieval**, **properties based retrieval** and **execution based retrieval**. Also this system provides a browse to allow user to browse library components based on the number of arguments.

<table>
<thead>
<tr>
<th>Theoretical works</th>
</tr>
</thead>
<tbody>
<tr>
<td>The system design</td>
</tr>
<tr>
<td>The features of the system</td>
</tr>
<tr>
<td>Software involved</td>
</tr>
<tr>
<td>Implementation:</td>
</tr>
<tr>
<td>WISER header</td>
</tr>
<tr>
<td>utilF interface</td>
</tr>
<tr>
<td>cel server and netscape interface</td>
</tr>
<tr>
<td>source code (type-based, semantics-based &amp; execution-based retrieval)</td>
</tr>
</tbody>
</table>

| The future of reuse |
A.3 WISER Browsing Page

![Browsing Library Components](image)

**Browsing Library Components**

(Two Arguments)

<table>
<thead>
<tr>
<th>Components Name</th>
<th>Components Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>inv2</code></td>
<td>(s-&gt;s-&gt;s)</td>
</tr>
<tr>
<td><code>mat2</code></td>
<td>(s-&gt;s-&gt;s)</td>
</tr>
<tr>
<td><code>insert</code></td>
<td>(a)-&gt;[a]</td>
</tr>
<tr>
<td><code>repeat</code></td>
<td>(a)-&gt;[a]+[a]</td>
</tr>
<tr>
<td><code>append</code></td>
<td>(a)-&gt;[a]-&gt;[a]</td>
</tr>
<tr>
<td><code>concat</code></td>
<td>([num]-&gt;[num]-&gt;[num])</td>
</tr>
<tr>
<td><code>subtr</code></td>
<td>(num-&gt;num-&gt;num)</td>
</tr>
<tr>
<td><code>add</code></td>
<td>(num-&gt;num-&gt;num)</td>
</tr>
<tr>
<td><code>mod</code></td>
<td>(num-&gt;num-&gt;num)</td>
</tr>
<tr>
<td><code>power</code></td>
<td>(num-&gt;num-&gt;num)</td>
</tr>
<tr>
<td><code>ldiv</code></td>
<td>(num-&gt;num-&gt;num)</td>
</tr>
<tr>
<td><code>round</code></td>
<td>(num-&gt;num-&gt;num)</td>
</tr>
<tr>
<td><code>isnum</code></td>
<td>(num-&gt;num-&gt;num)</td>
</tr>
<tr>
<td><code>isnum</code></td>
<td>(num-&gt;num-&gt;num)</td>
</tr>
<tr>
<td><code>isnum</code></td>
<td>(num-&gt;num-&gt;num)</td>
</tr>
</tbody>
</table>

Figure 56 WISER Browsing Page
A.4 WISER Retrieval Page

![Image of WISER Retrieval Page]

**Information of Arguments Type**

Please provide the type of arguments and the result type of components:

- **Type of Argument 1:**
- **Type of Argument 2:**
- **Return Type:**

- Type Retrieval
- Semantics Retrieval
- Execution Based Retrieval

[Submit Query] Submit your query [Reset] Clear Form.

Figure 57 WISER Retrieval Page
A.5 Insertion Page

Insertion of Components

Thanks for posting your code to us.
Note: The insertion must be done by system administrator. It may take some time.
Your code will be examined and then inserted into database.

I
Type of Argument1

I
Type of Argument2

I
Return Type

I
Function Name

Enter the source code: Enter the precondition:

Submit Query Submit your query. Reset Clear Form.
### Type Based Retrieval Result

<table>
<thead>
<tr>
<th>Components Name</th>
<th>Components Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abstract</code></td>
<td>(num -&gt; num -&gt; num)</td>
</tr>
<tr>
<td><code>add</code></td>
<td>(num -&gt; num -&gt; num)</td>
</tr>
<tr>
<td><code>divide</code></td>
<td>(num -&gt; num -&gt; num)</td>
</tr>
<tr>
<td><code>power</code></td>
<td>(num -&gt; num -&gt; num)</td>
</tr>
<tr>
<td><code>gcd</code></td>
<td>(num -&gt; num -&gt; num)</td>
</tr>
<tr>
<td><code>times</code></td>
<td>(num -&gt; num -&gt; num)</td>
</tr>
<tr>
<td><code>perm</code></td>
<td>(num -&gt; num -&gt; num)</td>
</tr>
</tbody>
</table>

Figure 59  WISER Retrieval Based on Type (Exact Match Components)
A.7 General Functions (Type retrieval)

![More General Components (type)](image)

<table>
<thead>
<tr>
<th>Components Name</th>
<th>Components Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>min1</td>
<td>(b+3a+3b)</td>
</tr>
<tr>
<td>max2</td>
<td>(a+2b+2a)</td>
</tr>
<tr>
<td>const</td>
<td>(a+b+2a)</td>
</tr>
</tbody>
</table>

Figure 60 WISER General Match Components Pages
### A.8 Freezing Argument Functions (Type retrieval)

**Freezing Arguments Components**

<table>
<thead>
<tr>
<th>Components Name</th>
<th>Components Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtract</td>
<td>(num -&gt; num -&gt; num)</td>
</tr>
<tr>
<td>add</td>
<td>(num -&gt; num -&gt; num)</td>
</tr>
<tr>
<td>divide</td>
<td>(num -&gt; num -&gt; num)</td>
</tr>
<tr>
<td>power</td>
<td>(num -&gt; num -&gt; num)</td>
</tr>
<tr>
<td>rcall</td>
<td>(num -&gt; num -&gt; num)</td>
</tr>
<tr>
<td>times</td>
<td>(num -&gt; num -&gt; num)</td>
</tr>
<tr>
<td>term</td>
<td>(num -&gt; num -&gt; num)</td>
</tr>
<tr>
<td>min2</td>
<td>(a -&gt; a -&gt; a)</td>
</tr>
<tr>
<td>max2</td>
<td>(a -&gt; a -&gt; a)</td>
</tr>
<tr>
<td>const</td>
<td>(a -&gt; b -&gt; a)</td>
</tr>
<tr>
<td>indexlen</td>
<td>([s] -&gt; num -&gt; a)</td>
</tr>
<tr>
<td>foldr</td>
<td>((a -&gt; b -&gt; b) -&gt; b -&gt; [a] -&gt; b)</td>
</tr>
<tr>
<td>foldl</td>
<td>((a -&gt; b -&gt; a) -&gt; a -&gt; [b] -&gt; a)</td>
</tr>
</tbody>
</table>

Figure 61 WISER Freezing Arguments Components Page
A.9 Semantics Based Retrieval Page

![Semantics Based Retrieval Page](image)

Figure 62 WISER Semantics Properties Based Retrieval Page
A.10 Execution Based Retrieval Page

Execution Based Retrieval Page

Please select testing method and submit your query:

◊ User Test ◊ System Test [Submit Query]

<table>
<thead>
<tr>
<th>Components Name</th>
<th>Components Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtract</td>
<td>(num &gt; num &gt; num)</td>
</tr>
<tr>
<td>add</td>
<td>(num &gt; num &gt; num)</td>
</tr>
<tr>
<td>divide</td>
<td>(num &gt; num &gt; num)</td>
</tr>
<tr>
<td>power</td>
<td>(num &gt; num &gt; num)</td>
</tr>
<tr>
<td>gcd</td>
<td>(num &gt; num &gt; num)</td>
</tr>
<tr>
<td>times</td>
<td>(num &gt; num &gt; num)</td>
</tr>
<tr>
<td>perm</td>
<td>(num &gt; num &gt; num)</td>
</tr>
<tr>
<td>min2</td>
<td>(a &gt; a &gt; a)</td>
</tr>
<tr>
<td>max2</td>
<td>(a &gt; a &gt; a)</td>
</tr>
<tr>
<td>const</td>
<td>(a &gt; b &gt; a)</td>
</tr>
</tbody>
</table>

Figure 63 WSIER Execution Based Retrieval Page
A.11 User Testing Page

![User Testing Page]

Please provide test data and the value of result:

<table>
<thead>
<tr>
<th>Value of Argument1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of Argument2</td>
</tr>
<tr>
<td>Value of Result</td>
</tr>
</tbody>
</table>

Submit Query Submit your query. Reset Clear Form.

Figure 64 WISER User Testing Page
A.12 System Testing Page

System Testing Page

Please provide the value of result for following test data:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \xi )</td>
</tr>
<tr>
<td>1</td>
<td>( -\xi )</td>
</tr>
<tr>
<td>2</td>
<td>( -\xi )</td>
</tr>
</tbody>
</table>

Submit Query   Submit your query. Reset   Clear form

Figure 65 WISER System Testing Page
A.13 Execution Based Retrieval Result Page

Execution Based Retrieval Results Page

<table>
<thead>
<tr>
<th>User Testing</th>
<th>System Testing</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Components Name</th>
<th>Components Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>(a-&gt;b-&gt;a)</td>
</tr>
</tbody>
</table>

Figure 66 WISER Execution Based Retrieval Result Pages
VITA AUCTORIS

Ping Bai was born in Beijing, China. She graduated from High School in 1979. From there she went on to the Beijing Computer Institute in China where she obtained a B.Sc. in Computer Science in 1983. She is currently a candidate for the Master's degree in Computer Science at the University of Windsor and hopes to graduate in the Spring of 1996.