1998

Retrieving function components from a reuse library.

Qiuyan An
University of Windsor

Retrieving Function Components From a Reuse Library

by
Qiuyan An

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

Windsor, Ontario, Canada
1998

0-612-52505-8

APPROVED BY:

[illegible]

Dr. Young Gil Park – Supervisor (*School of Computer Science*)

[illegible]

Dr. Subir Bandyopadhyay – Department Reader (*School of Computer Science)*

[illegible]

Dr. K. Y. Fung – External Reader (Dept. of Math. & Statistics)

Abstract

Increasing software development productivity is not enough to close the gap between the software demands in industry and software that can be provided in practice. Software reuse is claimed to be the only realistic approach to solve this problem [17]. As the reusable components are growing, we are faced the challenge of modulating, structuring and storing these components into reuse library so as to achieve faster and effective retrieval for reuse.

In this thesis we investigate an approach of structuring a function library and an efficient type-based retrieval method based on this structured library. In searching for functions, the trivial difference of the argument order of a function is disregarded. The library structuring is based on component grouping and component linking based on reusability relations among components. A prototype system **WISER** was also developed as a tool for achieving exact matched retrieval as well as relaxed matched retrieval. **WISER** also allows users to browse the structured library as an aid in finding potentially reusable components.

To my parents
my brother and sister
my husband Xiaobo and my son David

Acknowledgements

I would like to acknowledge the support and guidance provided by Dr. Y. Park. whose time. dedication and effort has contributed in guiding me through this thesis. Without his patience and guidance it would have been impossible to complete this thesis in this time frame. I am grateful to Dr. Subir for proving valuable suggestions and comments that aided in my progress. I would also like to thank Dr. K.Y. Fung for being willing to serve as my external reader.

Special thanks to my husband Xiaobo for his encouragement and support and my lovely son David for the relaxation time he gave me.

Last but not least. I would like to thank all my colleagues in the grad lab for providing a friendly atmosphere throughout my Master program.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Software Reuse

Software reuse was formally introduced in the late 1960's attributed to the "software crisis", which is still the concern in today's industry. Software reuse is defined as the "process of using existing software artifacts rather than building them from scratch" [1]. The "artifacts" includes every aspect of software lifecycle, documentation, specification, analysis, and source code.

Basically, the reuse-oriented software development process is composed of three phases: storage phase, retrieval phase and adaptation phase [14]. Figure 1 shows the process of reuse-oriented software development.

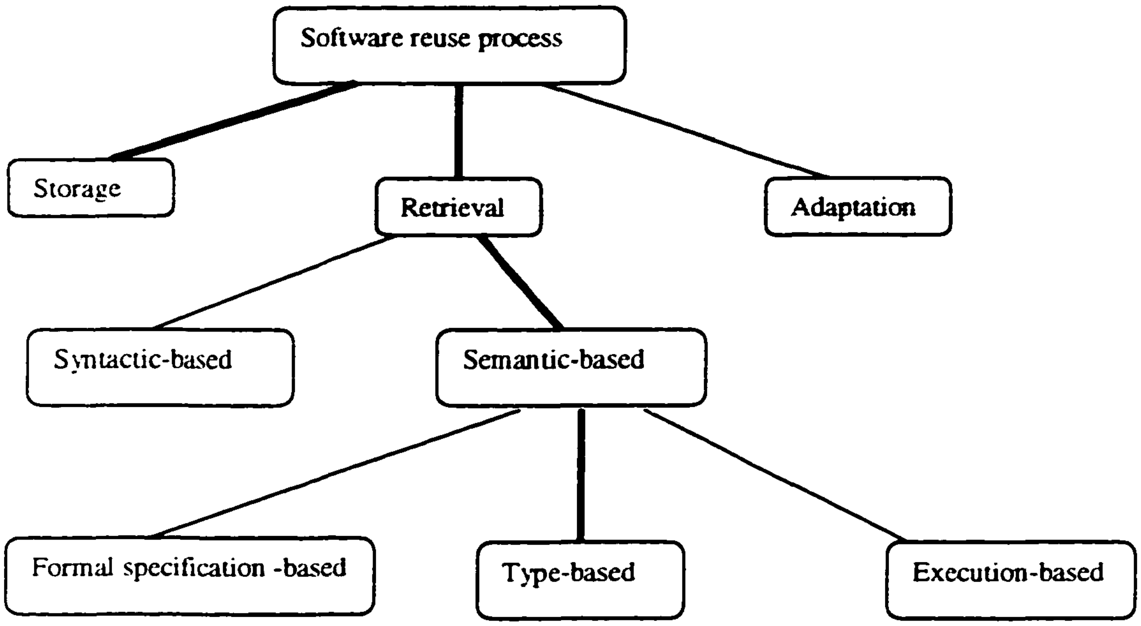
In storage phase, the software developer is responsible for putting the reusable components into a library, which will be used in the next phase. As the library grows, the issue of storing components becomes more serious. The software developers need to understand not only each component in the library but also the relationships between these components in order to store the components in a logically structured way. An abstraction over the component features is required.

In the retrieval phase, the library users try to find the desired component(s) from the library by applying his/her query. If the exact match can not be achieved, a potentially reusable relaxed match may be found. After some minor modifications on the retrieval result, the user can reuse it in his own application.

There are two types of approach for retrieving reusable components from a library. One is termed "**syntactic-based**" retrieval, another is called "**semantic-based**" retrieval. Syntactic-based retrieval deals with the syntax of the components. Keyword-based retrieval and facet-based retrieval belong to this type [19, 20]. Syntactic-based retrieval is easy to use and is very popular in today's software reuse market. The disadvantage of syntactic-based retrieval is that the users must have a knowledge base of the reuse source and the keyword set. Semantic-based retrieval deals with the meaning of the reusable components. It can be further divided into three categories: formal specification-based retrieval, execution-based retrieval and type-based retrieval. Formal specification-based retrieval makes use of specification language [2, 3, 5, 6, 7,18], which is proved to be ideal but hard to implement. Execution-based retrieval and type-based retrieval can be seen as an approximation of formal specification-based retrieval. Execution-based retrieval deals with the retrieval of executable components [4, 16]. The user provides the sample input and output either manually or systematically, the retrieval system then executes the components with the input and comparing the retrieval results with the output provided by the user. Execution-based retrieval can provide precise query result, but it is an expensive retrieval method due to the real execution on every related component. Type-based retrieval uses the component type as search key to query the library [9, 10, 11, 12, 13, 15]. Type is the specification over components, there are maybe many components in the library sharing the same type, so the retrieval result based on type is not precise, but it can serve as a filter and give a big cut over the unrelated components. Type-based retrieval approach can be integrated with other approaches such as execution-based retrieval to achieve more precise result.

The final phase deals with the adaptation of the components.

Figure1 Reuse-oriented Software Development Process



1.2 Functional Programming

As software becomes more complex, it is more important to structure it well. Well-structured software is easy to write, easy to debug, and provides a collection of modules that can be reused to reduce future programming costs. Functional programming has come of age over the last fifteen years. A variety of robust and efficient implementations of functional languages have been developed. A functional language is taught in many Universities as the first programming language.

Functional programming is based on the simplest of models, namely that of finding the value of an expression. Function programming consists of many built-in functions like +, -, which we can use to form expressions. We define functions by means of equations, like

addD x y = 2*(x + y)

which we use to calculate the value of an expression like ***addD 2 (add 3 4)***. We calculate the value in this procedure:

addD 2 (addD 3 4)

= 2 * (2 + (addD 3 4))

= 2 * (2 + 2*(3 + 4))

= 32

On top of this simple model we can build a variety of facilities, which can give functional programming its distinctive flavor. These include higher-order functions, whose arguments and results are themselves functions; polymorphism, which allows a single definition to apply simultaneously to a collection of types; and infinite data structures which are used to model a variety of data objects.

Miranda language is used in this thesis. Miranda language also has support for large-scale programming, including user-defined algebraic types, such as lists, trees and so on; abstract types and modules. These contribute to separating complex tasks into sub-tasks, making the components of systems independent of each other, as well as support software reuse.

1.2.1 Higher-Order functions

"A function which takes a function as argument, or delivers one as result, is called a higher-order function" [8]. The functions in function programming environment are the first class citizens and can be both passed as parameters and returned as results. Higher-order functions promote software reuse in functional programming environment because they allow partial parameterization.

In Miranda every function of two or more arguments is actually a higher-order function. For example, ***foldr*** is a higher-order function, it is defined by

```
foldr op k [] = k
foldr op k (a:x) = op a (foldr op k x)
```

All the standard list processing functions can be obtained by partially parameterising ***foldr***. Here are some examples:

```
product = foldr (*) 1
reverse = foldr postfix []
                where postfix a x = x ++ [a]
sum = foldr (+) 0
```

Let's see another example. "***member*** *is a library function such that* "***member x a***" tests if the list **x** contains the element ***a*** , this function returns ***True*** or ***False*** as appropriate. By partially parameterising ***member*** we can derive many useful predicates, such as:

vowel = member ['a', 'e', 'i', 'o', 'u']

month = member ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

1.2.2 Polymorphic Functions

Functional languages are strong typed, that is, every element of an expression and sub-expression has a type, which can be deduced at compiled time. Thus any expression which can not be assigned a sensible type is regarded as illegal and is rejected before evaluation. Strong typing does not required the explicit type declaration of functions Type can be inferred automatically.

Basic or primitive types are predefined and their values are built into the evaluator. In Miranda, there are three primitive types, called ***num***, ***bool***, and ***char***. Integer and floating point numbers share the same type ***num***, the distinction between integer and floating point numbers is handled at run time. There are two values of type ***bool***, called *True* and *False*. The type ***char*** comprises the ASCI character set.

There are also composite types in functional language. Three type constructors are used to form these types. "->" is used to form function type; "*(*, *)*" is used to form pair type; "*[ ]*" is used to form list type. For example, list a linearly ordered collection of values that can have an infinite number of elements inside it but they must all be in the same type, For example:

[1, 2, 3] which is of type ***[num]***

[False, True] which is of type ***[bool]***

[[1], [2], [3]] which is of type ***[[num]]***

Table 1 gives an example on type constructor

Table 1 Type constructor

Type	Description
a -> b	Function with argument type ***a*** and return type ***b***
(a, b)	Pair with left component type ***a*** and right component type ***b***
[a]	List containing elements of type ***a***

The variables ***a, b, c*** ... can be used as type variables, which means they are unknown types. In other words, one type variable can represent many types at the same time.

In summary, a type is either a type variable, a primitive type or a composite type.

Types containing no type variables are called **monomorphic types**. Table 2 gives us an example of **monomorphic type**.

Table 2 Example on Monomorphic type

avgr
Find the average of the list of numbers
avgr :: [num] -> num
avgr r [ ] = 0
avgr [a] = a
avgr a = sum a /lenlist a

Types containing one or more type variables are called **polymorphic types**. Table 3 shows us a **polymorphic function**. In **polymorphic function**, type variables can be replaced by any types. For example, we can use primitive type ***num*** to replace ***a*** in the function ***reverse***, then by applying the "***reverse***" function to a list of numbers, say *[1, 2, 3]*, we get the result ***[3, 2, 1]***. That is: ***reverse[1, 2, 3] = [3, 2, 1]***. If we replace ***a*** with ***char***, then for the sample input ***[a, b, c, d]***, we got the output ***[d, c, b, a]***. That is ***reverse [a, b, c, d] = [d, c, b, a]***. This example highlights the fact: Functions with **polymorphic types** can perform the same operation on some different data types. Another point is that the same type variable in a particular function stands for the same type at one time. For example, in function ***reverse***, you can not replace one type variable ***a*** with ***num*** and another with ***char***. This conforms to the unification theory. Type variable should be replaced unifiably to get an instance type.

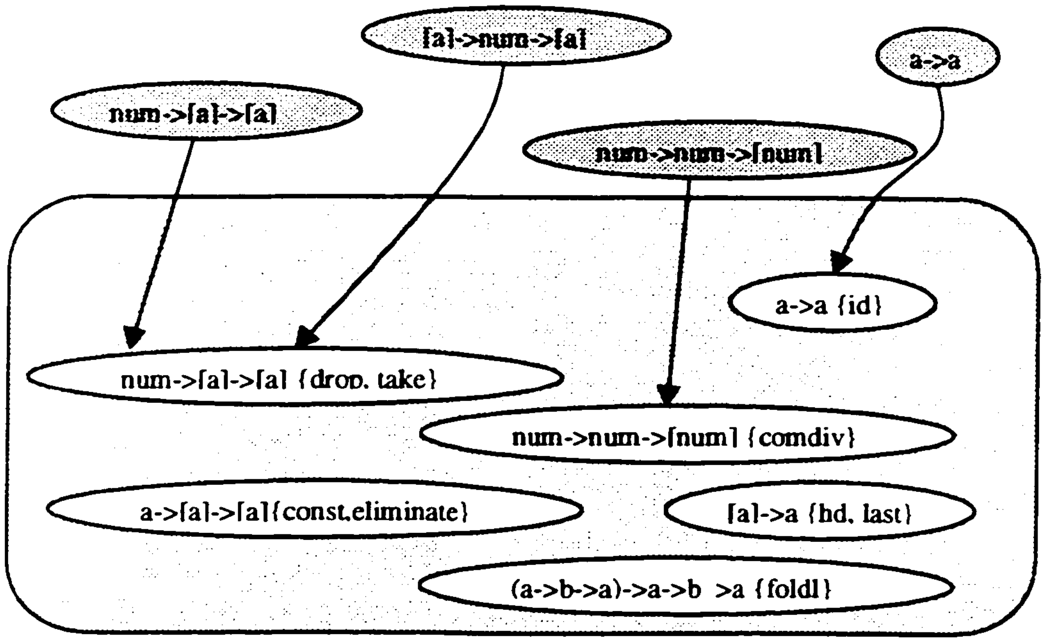
Table 3 Example on polymorphic type

reverse
Reverse a finite list
reverse :: [a] -> [a]
reverse = foldl prefix[ ]
where prefix xs x = x : xs

1.3 Type-based Retrieval of Functional Components

Most traditional libraries in functional programming are alphabetical or coarsely sorted by subject matter. They are tedious to search. Sometimes the library can be searched by specification, but even library functions were formally specified, it would not be possible to decide whether two specifications are equivalent or not. Based on this observation, we can choose type as search key in the retrieval, because type can be seen as an approximation of using specification as the search key and it is decidable [7]. Figure 2 shows us the use of types as search key.

Figure 2 The use of types as search key



One obvious problem in the type-based retrieval is that functions of more than one argument might be written with the arguments listed in any order. From the software developer's view, when he inserts the components in the library, he should have the flexibility to arrange the arguments of a component in any order without reformulating the arguments in exactly the same order as previous inserted components of the same type. But even he did so, it is still helpless to the user. Because there is no communication between the developer and the user about how the argument order of a component in library is arranged. From the user's view, a library user, who is looking for a particular function, should be able to specify the number of arguments, the input argument types and the output type of the function, but he has no way of knowing which choice of argument order is used in the library. Therefore, when searching a library, we don't simply want to see all functions of a given type, but all functions that could have this type if the order of the arguments is changed. In other words, we want to retrieve a set of functions which have the same argument number, same argument types and return type, the trivial difference of the order of arguments should be disregarded. Here is an example from Figure 2. Suppose we are looking for a two-argument function, what we know is that one argument type is ***num***, another argument type is ***[a]***, the return type is ***[a]***. The order of arguments is unknown. Then there are two possible types for this function:

num -> [a] -> [a]

[a] -> num -> [a]

All the functions in library with both of the types listed above should be retrieved, that is, both ***drop*** and ***take*** are the candidates for reuse.

Based on this idea, we'll define a type called **set_type**, which can denote all the function types without distinguishing the difference of the argument order.

The orders of argument types increases rapidly if the function has more than two arguments. In the worst case, say each argument carries a different type from each other, if we have ***n*** arguments, the possible function types should be ***n!***.

We need an algorithm of comparing the relationships between types, precisely speaking, an algorithm for comparing two function **set_types**. When you query the library with a giving function type, you got all candidates that have this **set_type**. This algorithm will be stated in **2.1**. Here is the strategy for type-based retrieval. When using type-based retrieval, the system does not search every component in the library. Type information of the components are stored in several text files. When a retrieval is requested, the system will search these text files and check if any match exists.

1.4 Overview of the Thesis

1.4.1 Motivation

Reuse-oriented development environment is claimed to be the healthy direction for software technology development. Software reuse can increase software quality and reliability because the reusable software components have been used and tested before. Software reuse can increase software productivity. Software reuse can also shorten the software release time to market, reduce maintenance cost [1, 14].

The fast growth of the Internet makes the research and collaboration on software reuse using the World Wide Web especially encouraging. Internet has created a market potential that much bigger than any one before.

We are facing the challenge of storing the large amount of software components into the reuse library in an efficient way so as to achieve effective retrieval.

There are two problems in the current reuse area in functional programming environment. The traditional approaches to reuse are basically relying on sequential searches to find suitable candidates [10, 11, 12], this leads to the problem of having to search the library many times before finding the candidate for reuse, especially in the case of relaxed retrieval, because in this case the library user needs to reformulate his query, there is no links between components in the library. The components are only listed sequentially. Another problem, as proposed in **1.3**, is the argument order problem [9, 15].

So far few work has been done to solve both of the problems. The work done in this thesis is to explore a practical method to modulate the function components based on type information by disregarding the difference of argument order and store them into a reuse library in order to achieve an effective retrieval.

1.4.2 The Objective of This Thesis

The objective of this thesis work is to investigate the use of a structured software library whose components are modulated by the type information. In particular, this thesis will focus on the first two phases of reuse-oriented software development process: the storage phase and the retrieval phase.

A prototype system called **WISER** is also designed and implemented for **Miranda** programming environment. This system provides library users with a friendly interface. This library allows users to browse and retrieve the components and can also provide "help" information for inexperienced users.

1.4.3 Thesis Statements

The thesis will defend the following statements:

1. Investigating of a type-based approach of modulating and storing software components in a library by disregarding the difference of the argument order and retrieving the components based on the reusable relations among the components.
2. Implementation of a prototype system based on our approach for functional programming environment in **Miranda**.
3. This system can be accessible from remote area through World Wide Web under client/server model.

1.4.4 Organization of the thesis

This thesis is organized into five chapters:

Chapter 1 gives a brief overview of the background knowledge involved in this thesis work, which includes software reuse, functional programming, type-based retrieval and an overview of the thesis.

Chapter 2 explains the details of the component module, the structuring of the library which entails the various relations that exist among the component modules. Also the structure of the record is explained.

Chapter 3 describes the detailed design and implementation of the prototype system called **WISER**.

Chapter 4 describes **WISER**'s interfaces and give some examples on how to use **WISER**.

Chapter 5 highlights the conclusions along with the future work.

Chapter 2 A Structured Approach to Type-based Retrieval

2.1 Comparison between Types

Formal specification characterizes the functionality of the components with well-defined syntax. it is ideal for the retrieval of reusable components. but it is generally hard to decide and implement. We have to choose a simple form of specification in terms of type. We summarize several reasons as why types can be used as search key in reuse-oriented program development. especially in functional programming environment:

1. Types are inherent in functions. they do not need to be derived or manufactured. They are available for use.
2. Type is a good filter. You can eliminate unrelated types quite easily.
3. Types are very important for reliable software. Strong typing can cause programming errors detected early at compile time.

Before proceeding. let's specify some name conventions in this thesis.

1. Lower case letter ***a***, ***b***, ***c***, ***d*** ,... are used for variable types.
2. For simplicity. if there are more than one type variable need to be used. we choose them by following the alphabet ascending order, that is, use ***a*** first. then ***b***. then ***c***...
   For example. we use ***a* -> *b*** to denote the most general type for two-argument functions rather than ***b* -> *a***. Another example. in a two-argument function. both the argument types are type variables which are not necessarily same. but the return type is same as the first argument type. By following this rule, we use function type ***a* -> *b* -> *a*** rather than ***b* -> *a* -> *b***.

"->" is used to construct function types. We need to define a new concept **set_type.** We will use symbol "***{*,* }***" to denote **set_type**, here "*****" stands for any argument type.

Set_type composes all the function types with the same number of arguments, same argument types, same return type but different order of arguments. For example, we have a two-argument function, one argument type is ***a***, the other argument type is ***num***, the return type is ***a***. We use ***{a, num} -> a*** (or ***{num, a} -> a***) to denote it rather than using both ***a -> num -> a*** and ***num -> a -> a***. As the number of arguments increases, there are more permutations on the order of argument types. We can see that **set_type** is a more compact, complete and powerful module than the single function type.

Let's take some time to explore the relations between **set_types**. The underline meaning of this comparison can be deployed in library search. One **set_type** denotes query type, one **set_type** denotes library identifiers (component module).

First, let's define the term ***general/specific***.

General/specific: If type ***E*** can be assigned to any ground types that is an instance of type ***A***, then type ***E*** is more specific than type ***A*** and ***A*** is more general than type ***E***.

Let's use ***T*** and ***T'*** to denote two function types. ***S*** and ***S'*** denote two function **set_types**. The functions have the same number of arguments, say ***n***

$T{::}\ A1 \rightarrow A2 \ldots An \rightarrow B$

$T'{::}\ A1' \rightarrow A2' \ldots An' \rightarrow B'$

$S = \{T1, T2, \ldots Tn\}$

$S' = \{T1', T2', \ldots Tn'\}$

Relation 1: Two function types are equal.

$T = T' \Leftrightarrow$ if for all $1 \leq i \leq n$

there exists ***Ai = Ai'***,

B = B'

then ***T = T'***

e.g. ***T:: a -> num -> bool***

T':: a -> num -> bool

Relation 2: ***T'*** is more general than ***T*** and ***T*** is more specific than ***T'***. Type ***T*** is an instance of ***T'*** if ***T*** can be obtained from ***T'*** by relevant linear consistent substitution of type for previous defined type variables that occurs in type ***T'***.

T < T' , ***T' > T*** ⇔ if for all $1 \leq i \leq n$

there exists ***Ai < Ai'*** *or* ***Ai = Ai'***,

B < Bi' *or* ***B = Bi'***,

but not ***T = T'***

then ***T < T', T'> T***

e.g. ***T :: [num] -> num -> bool***

T' :: [a] -> a -> bool

Relation 3: ***T*** and ***T'*** has no relation.

if not exist ***T = T', T < T', T > T'***

then ***T*** and ***T'*** has no relation

e.g. ***T:: [a] -> [a] -> a***

T':: [num] -> num -> bool

Relation 4: Two function **set_types** are equal. If ***S*** and ***S'*** contains the same set of argument types and same return type, then ***S = S'***.

e.g. ***S:: {[a], a} -> bool***

S':: {a, [a]} -> bool

Relation 5: ***S'*** is more general than ***S*** and ***S*** is more specific than ***S'***.

$S < S'$, $S' > S$ ⇔ if for $1 \leq i \leq n$, $1 \leq j \leq n$

there exists $Ti < Tj'$

then $S < S'$

e.g. ***S:: {num, [num]} -> bool***

S':: {[a], a} -> bool

Relation 6: *S* and ***S'*** has no relation.

if not exist $S = S'$, $S < S'$, $S > S'$

then ***S*** and ***S'*** has no relation

e.g. ***S:: {[a], a} -> a***

S':: {[a], a} -> num

We implement an algorithm based on the above definitions for comparing the relations between **set_types**. We only introduce the basic idea about the algorithm. See chapter 3 for detail. Basically there are four steps in the algorithm.

1. Compare two element types. The element type means the argument type and the return type. For example, a type variable is more general than any primitive type.
2. Compare two function types based on 1. If a function type can be derived by replacing the type variables in another function unifiably, this function is more specific than that function. For example, ***num -> [char] -> [char]*** can be obtained by replacing all the occurrences of type variable ***a in num -> [a] -> [a]***. Sometimes two functions have no relation. For example, ***a -> a -> a*** has no relation with ***a ->[a]->[a]***.

3. Compare one function type with one function **set_type**. This is done by comparing this function type with all the function types in this set_type until a relation is found or the search exhausts.
4. From **relation 4, 5, 6**, we can derive the relation for two function **set_type**.

2.2 Functions with More General Set_type

Assume the library is well designed and the user is not allowed to insert component into library. Then there are at least two reasons for retrieving functions with more general **set_type**:

1. If an exact match is found yielding functions that are of no significance to the user, then more general functions should be pursued.
2. Sometimes the user can not formulate the most general type for his required function, but instead he can give an instance of that function type, then the desired function would be more general than the query type. In this case, the library structure helps the user find his desired function.

For example, if the user searches a function which compares two numbers and gives the bigger one. If he uses ***{num, num} -> num*** as search key to query the library, he got the results "***add***", "***comb***", and "***gcd***". By investigating the source code of these functions, the user found that none of them can satisfy his requirement. Based on this exact match, the user can apply the general match and find the function "***max2***", which compares two elements and returns the bigger one. By replacing the type variable ***a*** with ***num***, the user got his desired function.

We use some strategies when we implement this algorithm. For example, when we compare two function types, we compare their return types first, if their return types have no relation, we can conclude that these two function types have no relation. Further comparison in this case is not necessary. This algorithm is therefore very efficient.

2.3 Functions with More Specific Set_type

In 2.2, we discuss the issue of specializing the library function to meet the user's desire. Here we discuss an opposite situation, generalizing the specific function to meet the user's requirement.

When the exact match or general match function(s) in the library still can not meet the user's requirement, there may be some specific functions useful. If the source code is written very well, the user can reuse them by simply generalizing the source code.

For example, if a user wants to find a function which is used to insert an element into a list of elements, the query type is ***{a, [a]} -> [a]***, by applying exact-match, none of the retrieval results can satisfy the user's desire. If he performs a specific match based on the exact match, he can get a function called "***numInsert***", which is for inserting a number into a list of numbers. By just modifying the type number to type variable ***a***, "***numInsert***" can be reused easily.

2.4 Functions with Extra-argument

When implementing a same purpose function, different programmer may give different type. For example, one program may implement the function "***numsort***" with type ***[num] -> [num]***, another programmer may use type ***{(num -> num -> bool), [num]} -> [num]***. So the extra-argument function has reuse value. We should apply it into our library design.

2.5 Structure of the Reuse Library

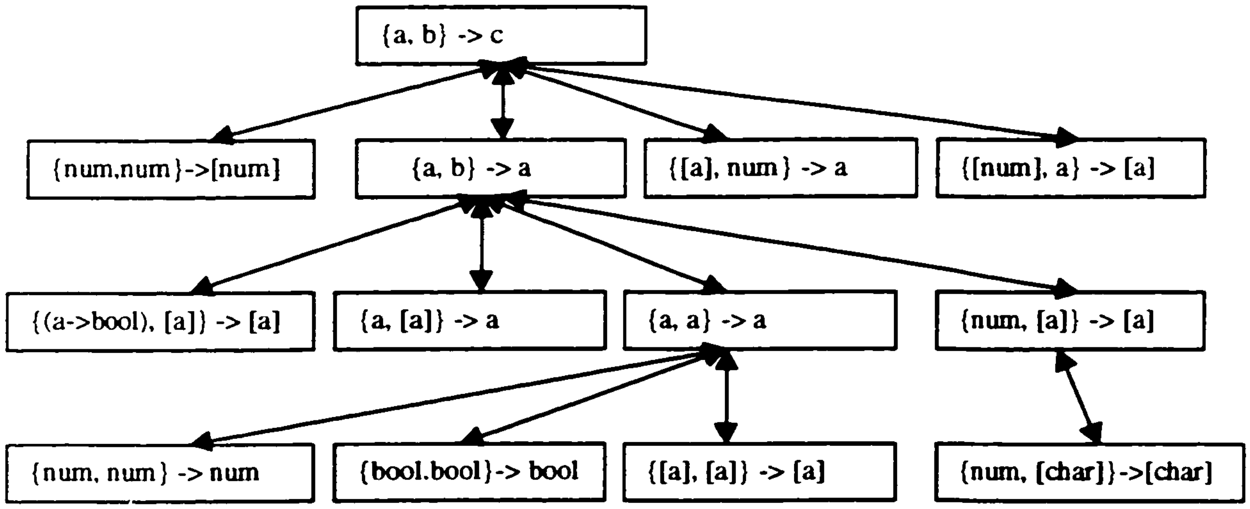
In the implementation of the reuse library, we virtually divided the whole library into sub-libraries based on the number of arguments for the functions. All the functions with the same number of arguments are in the same sub-library. To do this, we keep separate text file for each sub-library, the text files are used as the management system of server, all the relations among the components are reflected on the text files. All the functions (source code) are actually stored in somewhere on disk.

Based on the analysis of **2.1** to **2.4**, we can specify two categories of relations, named ***intra-library links*** and ***inter-library links***. (Here library means sub-library).

2.5.1 Intra-library Links

Intra-library links deal with general relation and specific relation between functions. Based on these relations and the **set_type** concept, we can construct a graph, which visually reflect the various relations and give you an intuitive feeling about the structure. Figure 2 is a sample graph.

Figure 3 Example of Intra-library Links



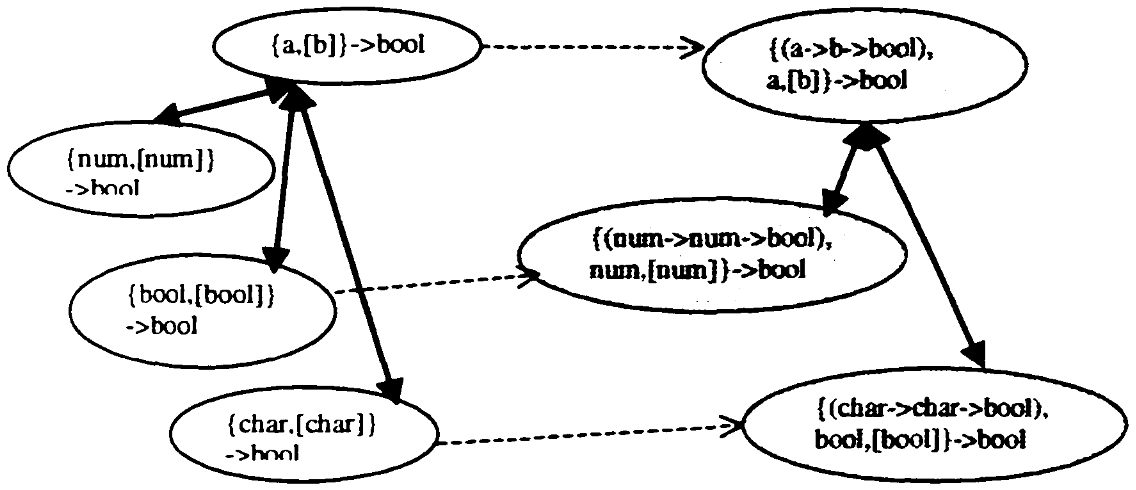
From the graph, we can have the following observations:

1. One node (module) may have more than one immediate parent and immediate child. For example, the node with **set_type** ***{a, b} -> a*** has one immediate parent ***{a, b} -> c*** and four immediate children ***{(a->bool), [a]} -> [a], {a, [a]} -> a, {a, a} -> a*** and ***{num, [a]} -> [a].***
2. Root node is the most general **set_type**, it has no parent.
3. Leaf node is the most specific node in the current path in the graph, it has no child for the time being. Some leaf node can not be specialized any more, like ***{num, num} -> num***, ***{bool, bool} -> bool*** and ***{num, [char]} -> [char].*** But some leaf node still have the potential to be specialized, like ***{[a], [a]} -> [a], a*** can be replaced by other types.
4. By navigating the graph, the user can obtain all the functions with general **set_type** and specific **set_type**. The search is not only restricted to the immediate ones.
5. The library is assumed to contain many components. In most of the case, the user can find the exact matched **set_type**. The further general match and specific match are all based on the exact match. If the exact match **set_type** is not in the library, we can not perform the flexible match retrieval. For security and maintenance reason, we do not allow the general user to have "write" and "modify" permission to the library. If this is allowed, the library will be messy and out of control.

2.5.2 Inter-library Links

Inter-library links reflect the extra-argument relations between components. Inter-library links provide us with one more option if our desired functions can not be obtained from previous efforts. Figure 4 shows us an example of inter-library relation. From the graph, we observe that the inter-library link is one-directional, it starts from the library with less arguments, ends with the library with one more argument.

Figure 4 Example of Inter-library links


{a,[b]}->bool
{(a->b->bool), a,[b]}->bool
{num,[num]} ->bool
{(num->num->bool), num,[num]}->bool
{bool,[bool]} ->bool
{char,[char]} ->bool
{(char->char->bool), bool,[bool]}->bool

2.5.3 Sample of structured Software Library

The reuse library is constructed by using both the intra-library relation and inter-library relation. The library is designed based on these relations so as to promote the automatic retrieval. In the case of exact match not satisfying the user's requirement, the user does not need to reformulate his query. General match, specific match and extra-argument match can be performed automatically based on the choice of the user.

Figure 5 is a sample of our software reuse library containing all kinds of links mentioned above.

2.6 Structure of a Record

Each sub-library information is kept in a separate text file. Because the library is dynamic changing as the insertion or deletion occurs, so we must use the random access file to keep information. In the implementation of random access file, we choose fixed-length record method (160 bytes per record). There are five fields in a record. They are set_type, function names, general links, specific links and extra-argument links. One record can be seen as a module which encapsulates the information about the five fields. Table 4 shows the structure of a record.

Figure 5 Sample of Our Software Library

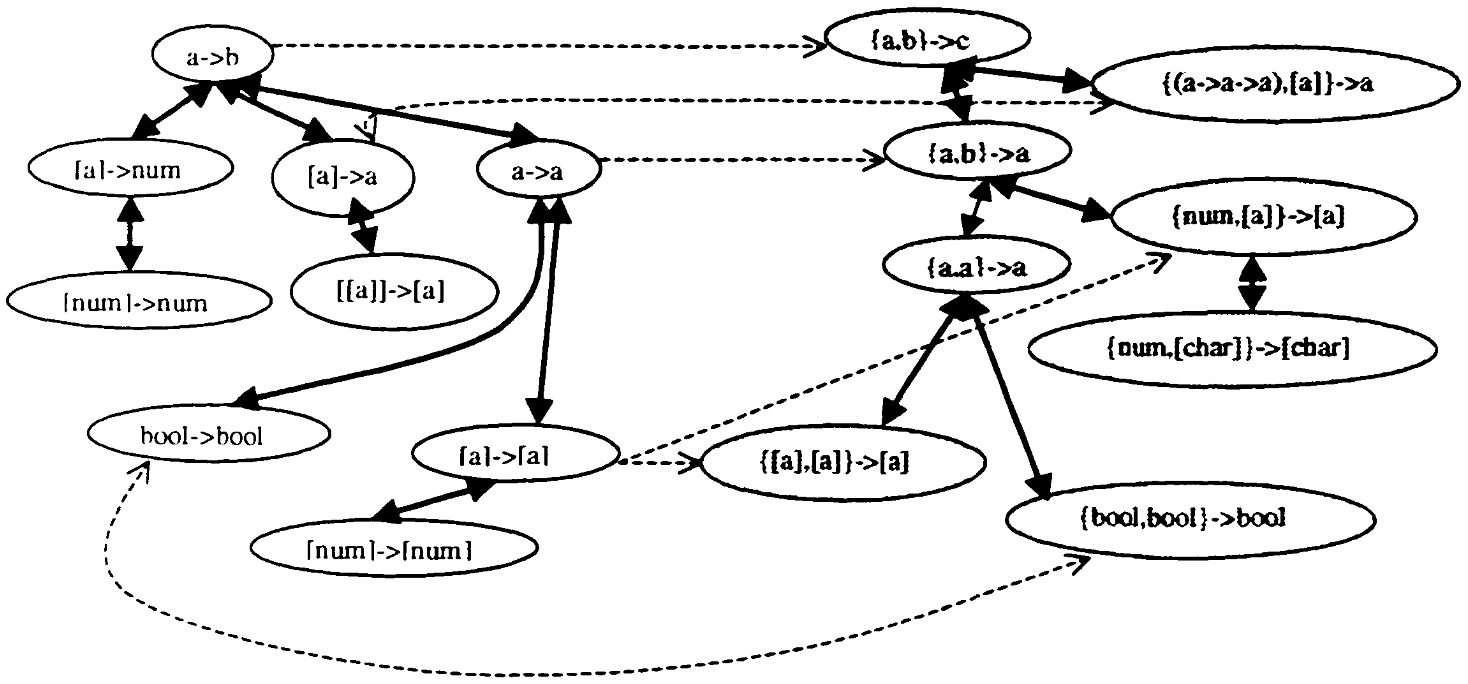


Table 4 Structure of a Record

Σ {FΣ} {GΣ} {SΣ} {EΣ}

Σ represents the set_type

FΣ represent all the functions with set_type Σ

GΣ represents the offsets of all the immediate more general set_type(s) of Σ

SΣ represents the offsets of all the immediate more specific set_type(s) of Σ

EΣ represents the offsets of all the immediate extra-argument set_type(s) of Σ

To gain a fully understanding of the record, let's see the following sample record:

{num\[a]\[a]}{drop\take}{160}{1920}{ }

We can deduct from this record that the functions "***drop***" and "***take***" has the **set_type** ***{num, [a]} -> [a]***. At offset 160, you can find a record whose **set_type** is more general than ***{num, [a]} -> [a]***. At offset 1920, you can find a record whose **set_type** is more specific than ***{num, [a]} -> [a]***. There is no extra-argument link for this record.

Chapter 3 A Prototype System WISER

A prototype system **WISER** is designed and implemented by using our approach. **WISER** stands for **WIndsor Software basE for Reuse**. It is an interactive software reuse system which is directed at the storage and retrieval phases of the reuse-oriented program development. It is based on client/server model. At the server side, the library developer can perform insertion, deletion, browsing and retrieval operations. At the client (user) side, the library users can perform browsing, retrieval, showAll operations, a "help" information is also provided for inexperienced users.

Reuse by **WISER** is on the source code level. When the user writes code in **Miranda**, he/she can query **WISER** to find desired functions. Programming by reuse takes many advantages than writing the programs from scratch.

All the features of **WISER** rely on the process of determining whether a query type falls into an existing **set_type** in library. The user is required to supply information about the number of arguments, the types of arguments and return type. The order of argument types of the function is not required.

This chapter covers the design of **WISER**, and focuses on the analysis of insertion, deletion, retrieval and browsing based on the structured library.

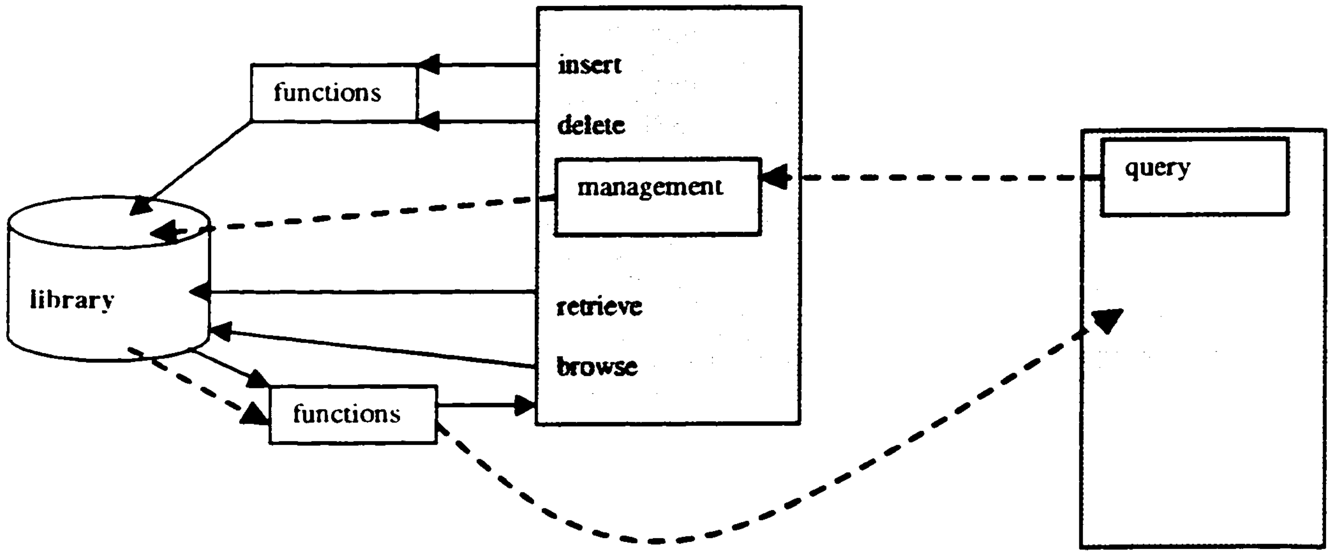
3.1 System Design

WISER is designed to help users in their program coding phase. Figure 6 is an overview of **WISER** structure.

We list some considerations during **WISER** design process:

1. Only the library developers have the "write" right to the components in the library. This means that insertion and deletion can only be done by the library developers.

Figure 6 Overview of WISER



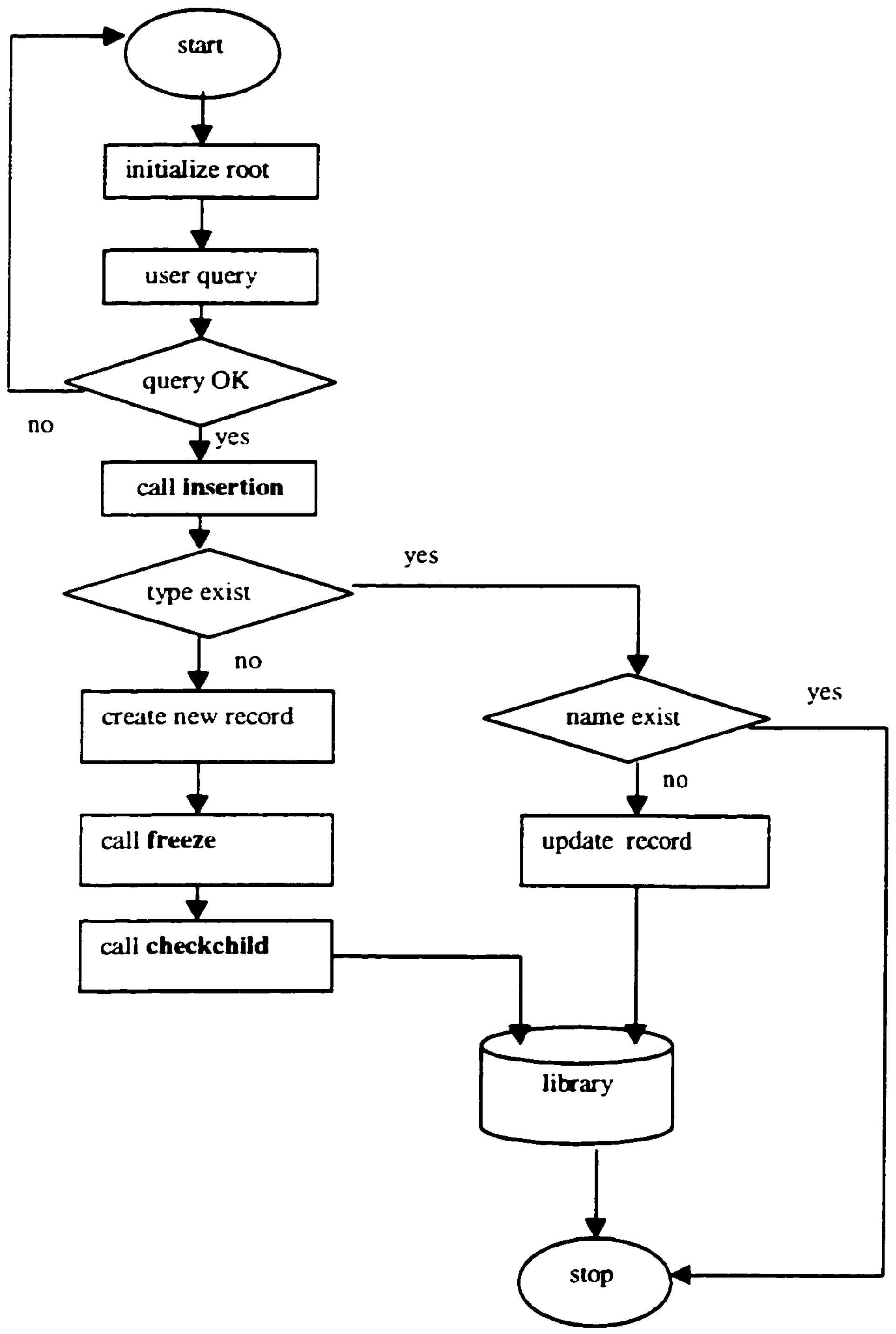
2. This system allows replacement of component. This is done by first deleting the old component and then inserting the new one.
3. Once the library is established, the concerns will be put on the automatic retrieval phase. **WISER** supports exact match as well as relaxed match (general match, specific match, extra-argument match), this greatly increases the potential reuse value of the components. The users have more options to get his/her desired functions.
4. A Browser is provided to overview the structured library as an aid in finding potentially reusable components.
5. "showAll" is a shortcut to view the list of components in a specified sub-library without providing type information.
6. A "help" is provided for facilitating inexperienced users.
7. The interface should be user-friendly.

3.2 Code Documentation

3.2.1 Insertion

Insertion is the most complex operation in **WISER** system design. Some conditions must be checked in order to insert the component into right position in the structure. Figure 7 is basic structure of insertion.

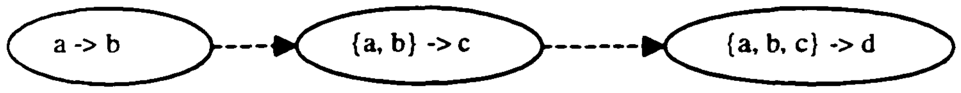
Figure 7 Structure of Insertion



According to this structure, the insertion starts with the library developer supplying the required information, which should include the number of arguments, the types of arguments and the function return type. The argument types can be given in any order. A confirmation is then submitted by the system in response to the query so that corrections can be made in case of errors in the query. Then the function is inserted into the library in accordance with its type. If there does not exist a **set_type** in the library that corresponds to the query type, a new node will be created. Some conditions must be checked to enforce correct insertion. If there is a type in library corresponding to the query type, a further check is done by the system to see if the function name already exists. If the function name does not exist, the record will be updated by adding the function name in the corresponding field of the record, the various links of this record keeps untouched. Otherwise, if the function name does exist in the library, no insertion is needed, and the insertion procedure is terminated.

The test involved in the insertion is to decide the relations between two **set_types**. To do this, we use the **relation 4, 5, 6** as explained in **2.1**. By using these relations we can proceed to insert function components into the reuse library correctly. Before any insertion performed, the root of each file for the structured library must be set. The root is a record containing the most general type of all records in the file. Furthermore, they are linked together based upon the extra-argument relation. Figure 8 is the library structure after the insertion of the root record.

Figure 8 Insertion of root records



Take node *{a,b}* **->** *c* as example, the record representing this node is ***{a\b\c}{ }{ }{ }{0}***. which means that this two-argument root node has no function name attached to it, it has no general and specific links and it has an extra-argument link pointing to the offset of 0 in the three-argument file.

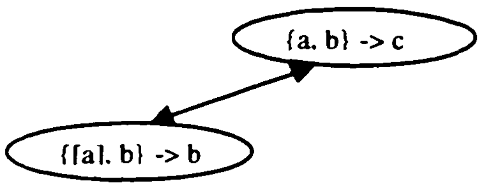
The insertion is in the form of records with fields as we discussed in **2.6**. Insertion consists of determining the insertion record's children, parents and extra-argument links as well as updating the various existing links.

To gain a better understanding to insertion, we give two examples in order to highlight all the techniques involved in insertion. In the first example, we will discuss insertion based on the polymorphic relations. In the second example, we have a chance to see how the extra-argument relation is deployed in insertion.

Example 1: We assume that the root node is the only node in the two-argument structure. We proceed to insert the type ***{[a], b} -> b*** into this structure. Figure 9 shows this two-argument structure. This is because:

{[a], b} -> b < {a, b} -> c

Figure 9 Insertion of *{[a], b} -> b* into two-argument structure

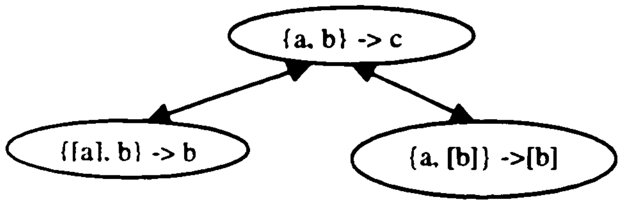


Next we insert the type ***{a, [b]} -> [b]*** into the structure yielding Figure 10. This is because:

{a, [b]} -> [b] < {a, b} -> c

{a, [b]} -> [b] has no relation with ***{[a], b} -> b***

Figure 10 Insertion of *{a, [b]} -> [b]* into two-argument structure



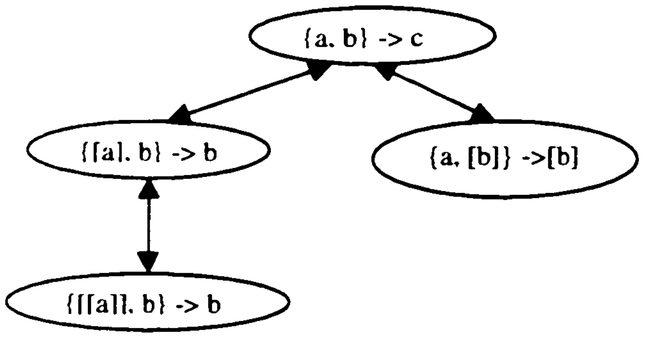
We continue insert ***{[[a]], b,} -> b*** giving figure 11. This is because:

{[[a]], b} -> b **<** ***{a, b} -> c***

{[[a]], b} -> b **<** ***{[a], b} -> b***

{[[a]], b} -> b has no relation with ***{a, [b]}-> [b]***

Figure 11 Insertion of *{[[a]], b} -> b* into two-argument structure



The insertion seems to be based upon instances alone and once an instance is found we link it as we did in Figure 9, 10, 11. However this is partially correct as shown in Figure 12 when we insert the type ***{[[a]], [b]} -> [b]***. The relevant links were formed because:

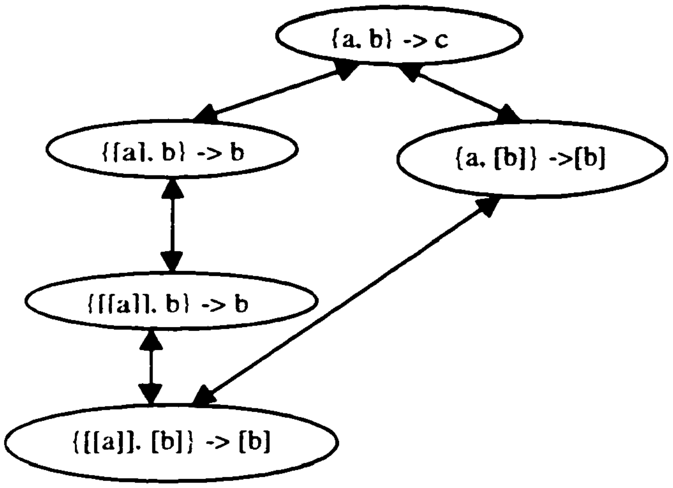
{[[a]], [b]} -> [b] **<** ***{a, b} -> c***

{[[a]], [b]} -> [b] < ***{a, [b]} -> [b]***

{[[a]], [b]} -> [b] < ***{[[a]], b} -> b***

{[[a]], [b]} -> [b] < ***{a, [b]} -> [b]***

Figure 12 Insertion of *{[[a]], [b]} -> [b]* into two-argument structure



So far we have shown that a node in the structure may have more than one parent. We need to search from left to right, from top to bottom to determine whether or not it has more parents. This search process is recursively performed. Part of our function **insertion** tests for these conditions mentioned above.

Suppose we have the structure as shown in Figure 13. We now insert type ***{a, a} -> a***. According to this structure we have:

{a, a} -> a < ***{a, b} -> c***

{a, a} -> a has no relation with ***{[a], b} -> b***

{a, a} -> a has no relation with ***{a, [b]} -> [b]***

Based on this analysis, we have the structure as shown in Figure 14. Unfortunately, this is not entirely correct since we know:

Figure 13 two-argument structure before insertion of *{a, a} -> a*

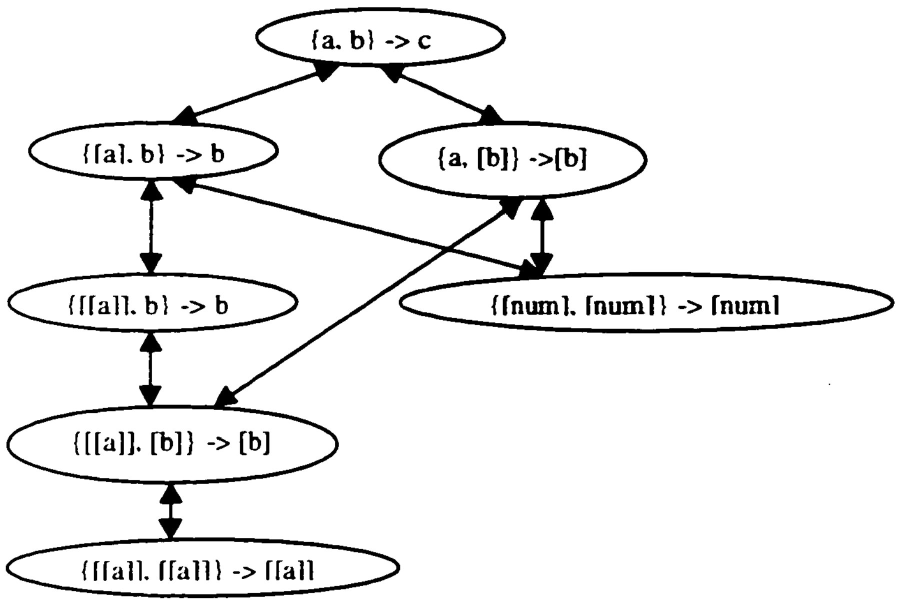
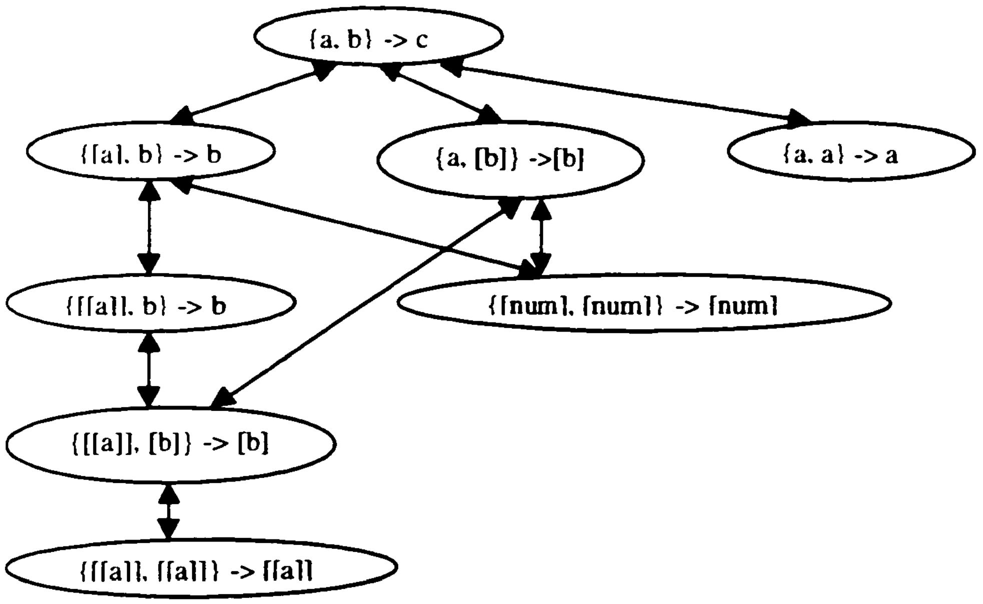


Figure 14 Incorrect insertion of the insertion of *{a, a} -> a into two-argument structure*

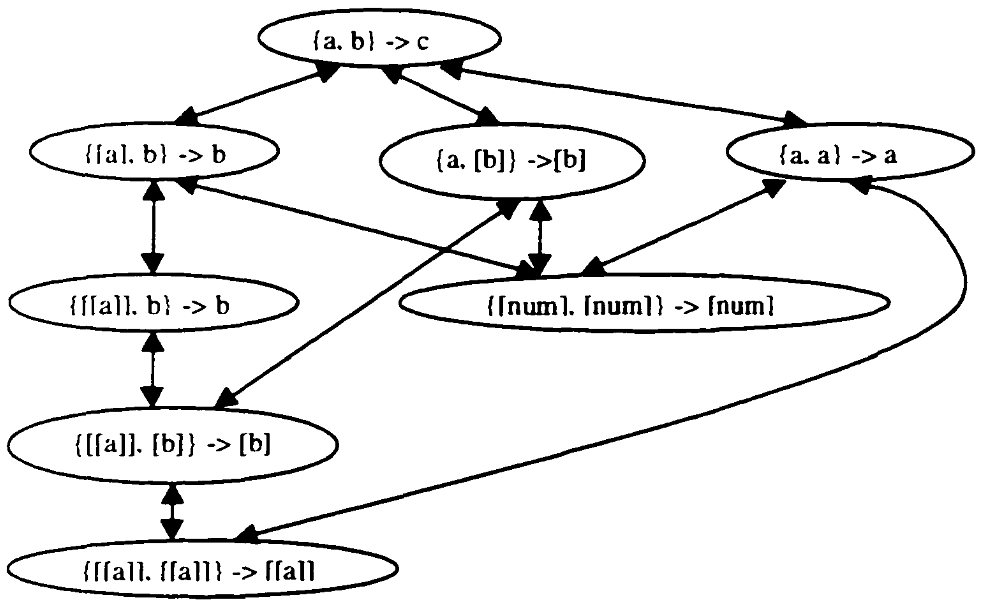


{a, a} -> a > {[num], [num]} -> [num]

{a, a} -> a > {[[a]], [[a]]} -> [[a]]

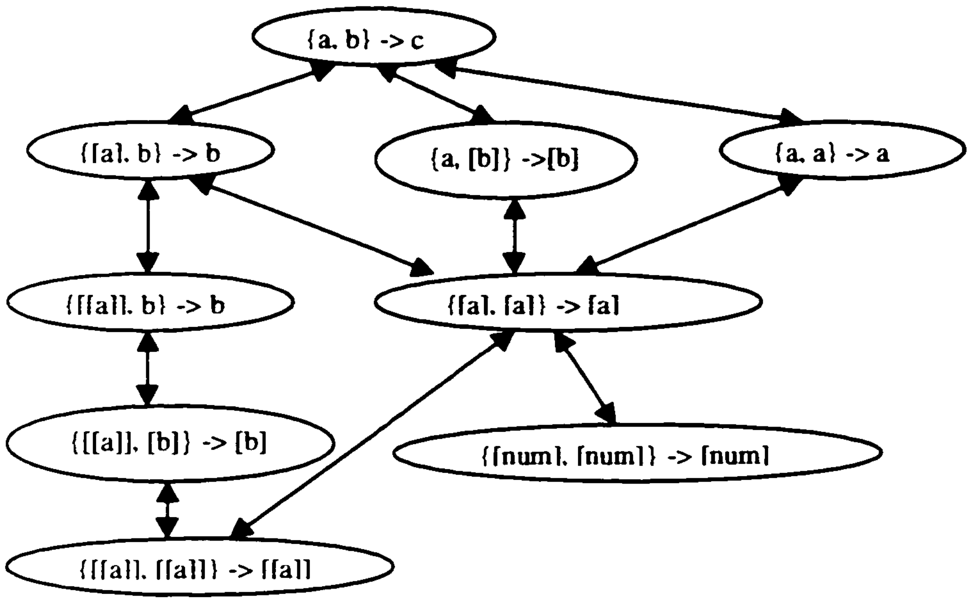
So we need go back and check almost all the existing nodes to see if it is an instance of the new inserted node. One node may have more than one child in the structure. Furthermore if a type is found to be an instance of the inserted one, it should be further checked to ensure its links does not exist already. Our function **checkchild** was used in testing this condition. Furthermore, once an instance is found none of its instances needs to be checked since they too will be instances of the inserted type, although not the immediately ones. Part of our function insertion tests this condition. Figure 15 shows the correct structure after insertion of ***{a, a} -> a***.

Figure 15 Correct insertion of *{a, a} -> a*



Continue to insert type ***{[a], [a]} -> [a]*** into the structure of Figure 15, we produce the structure of Figure 16.

Figure 16 Insertion of {[a], [a]} -> [a] into two-argument structure



From this graph, we observe that:

{[a], [a]} -> [a] < *{a, b} -> c*

{[a], [a]} -> [a] < *{[a], b} -> b*

{[a], [a]} -> [a] has no relation with ***{[[a]], b} -> b***

Which shows ***{[a], [a]} -> [a]*** is an instance of ***{[a], b} -> b***. Continue along we have:

{[a], [a]} -> [a] < ***{a, [b]} -> [b]***

{[a], [a]} -> [a] has no relation with ***{[[a]], [b]} -> [b]***

{[a], [a]} -> [a] > ***{[num], [num]} -> [num]***

Which shows that type ***{[a], [a]} -> [a]*** is a child of type ***{a, [b]} -> [b]*** and also ***{[a], [a]} -> [a]*** has a child of type ***{[num], [num]} -> [num]***. Continue along we have:

{[a], [a]} -> [a] < ***{a, a} -> a***

{[a], [a]} -> [a] > ***{[num], [num]} -> [num]***

{[a], [a]} -> [a] > ***{[[a]], [[a]]} -> [[a]]***

Which shows that type ***{[a], [a]} -> [a]*** is a child of type ***{a, a} -> a*** and type ***{[a], [a]} -> [a]*** has children of type ***{[num], [num]} -> [num]*** (already exists so ignored) ***{[[a]], [[a]]} -> [[a]]***. In this case, ***{[[a]], [[a]]} -> [[a]]*** takes all the children of ***{a, a} -> a***, but not takes all the children of ***{[a], b} -> b*** and ***{a, [b]} -> [b]***. The function **insertion** also tests for the children changing condition.

There is a final condition. When an exact match of type occurs, the function name is inserted into the record if it does not exist already. The function ***insertion*** is also used here.

Example 2:

Figure 17 Before insertion of *{[a], [a]} -> num* into the structured library

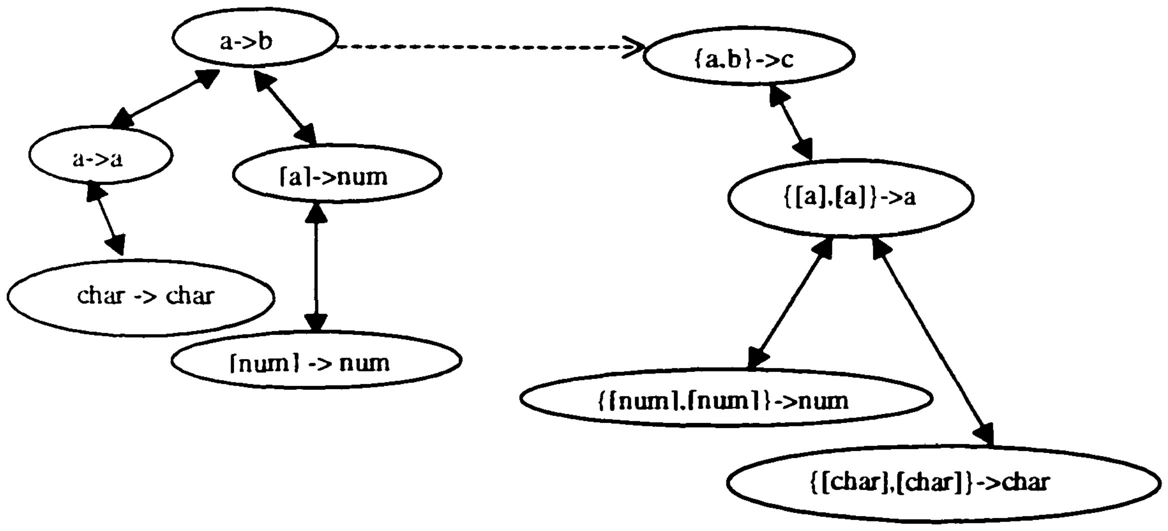
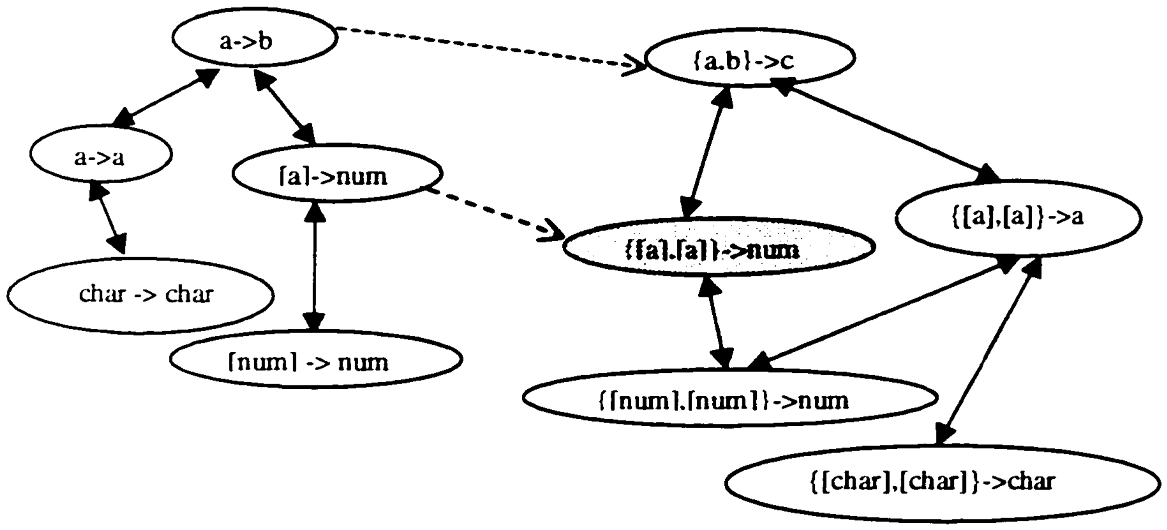


Figure 18 After insertion of *{[a], [a]} -> num* into the structured library



Assuming the current library structure is as Figure 17. We proceed to insert function of type *{[a], [a]}* -> ***num*** into this structure. Figure18 is the structure after insertion of *{[a], [a]}* -> ***num***. This is because:

{[a], [a]} -> ***num*** < ***{a, b}*** -> ***c***

{[a], [a]} -> ***num*** has no relation with ***{[a], [a]}*** -> ***a***

So we create a new node of type ***{[a], [a]}*** -> ***num***. Continue checking:

{[a], [a]} -> ***num*** > ***{[num], [num]}*** -> ***num***

{[a], [a]} -> ***num*** has no relation with ***{[char], [char]}*** -> ***char]***

Type *{[a], [a]}* -> ***num*** has a child of type ***{[num], [num]}*** -> ***num***. This is done by calling function **checkChild**. The related records affected by the insertion (***{a, b}*** -> ***c*** and ***{[num], [num]}*** -> ***num***)need to to updated at corresponding field.

By freezing one of the arguments in type ***{[a], [a]}*** -> ***num***, we get the new type ***[a]*** -> ***num***. We find this new type ***[a]*** -> ***num*** is in the #arg = 1 sub-library. That is, ***[a]*** -> ***num*** has an extra-argument match type ***{[a], [a]} -> num*** in #arg = 2 sub-library. Function **freeze** is used to create this extra-argument link. The record of type ***[a] -> num*** is updated at the extra-argument field.

From above analysis about the various conditions, we can conclude that insertion is a complex operation in the structured library, care must be taken when we perform a correct insertion. The insertion is best accomplished by using recursion. Type checking is the most important part in the insertion. We will describe the algorithm for checking relations between two function types. This **check_function** algorithm is heavily used in the **algorithm of insertion**.

Here is the algorithm for checking relations between two function types:

check_function(*f::T, f :: T'*)

Suppose ***f :: T = A1 -> A2 … -> An -> B***

f' :: T' = A1' -> A2' … -> An' -> B'

Step1: function **check_token(toke_type2, token_type2)** is used to decide the relation between ***Ai*** and ***Ai'***, ***B*** and ***B'***.

Table 5 Relations between two basic types

	num	***bool***	***char***	***[*]***	***(*, *)***	***a, b, c …***
num	=	no relation	no relation	no relation	no relation	<
bool	no relation	=	no relation	no relation	no relation	<
char	no relation	no relation	=	no relation	no relation	<
[*]	no relation	no relation	no relation	see note 2	no relation	<
(*, *)	no relation	no relation	no rėlation	no relation	see note 2	<
a, b, c …	>	>	>	>	>	see note 3

We have six kinds of basic types which can be used as argument types and return type except function type according to the discussion in **1.2.2**. Let's define all the relations between any two kinds of basic types. Table 5 shows the comparison results. Let's give the explanation about the table.

<u>note 1</u>: Any primitive type is equal to itself and more specific than any type variable, but has no relation with other primitive type, pair type or list type.

<u>note 2</u>: Both of list type and pair type have no relation with primitive types (***num***, ***bool***, **char**) and are more specific than type variables. They have no relation with each other. List type may or may not have relation with another list type, this can be decided by taking off the surrounding "*[*" and "*]*" and comparing the component type of the list using this algorithm recursively. For example, when we compare two list type ***[a]*** and ***[num]***, we take off the surrounding "*[*" and "*]*" and get the component type ***a*** and ***num***. We know ***a*** > ***num***, then we can conclude that ***[a]*** > ***[num]***. Table 6 are some examples of the comparison between two list types. The same comparison mechanism can be applied for the comparison of two pair types.

Table 6 Examples of the comparison between two list types

	[a]	***[num]***	***[[a]]***	***[(num, bool)]***
[a]	=	>	>	>
[num]	<	=	no relation	no relation

<u>note 3</u>: A type variable is always general than primitive type, list type and pair type and equal to itself. According to our name convention, if a particular type variable appears before another type variable in the alphabet list, then this type variable is more specific

than the later one. For example, type variable ***a*** is more specific than type variable ***b*** and *c*.

<u>**note 4**</u>: If the argument type or return type is a function type (higher-order function), we call function **check_function** to decide the relation. (See step 2).

Step 2: function **check_function(*f :: T, f':: T'*)** is used to decide the relation between *f* :: ***T*** and *f* :: ***T'***

The six basic types are the building blocks in constructing a function type. The comparison of function types is the process of comparing all the corresponding argument types or return tpes, as we described in **2.1**. We use some strategies to perform the function comparisons in order to reduce the checking time. For example, we can compare the return types first. If there is no relation between the return types, we can conclude that the two function types have no relation. The following is the description:

```
if (f::T == f'::T')          // string comparisom
   then f :: T = f' :: T'
else
   if B has no relation with B'
      then f :: T has no relation with f' :: T'
   else
      if B == B'
         flag = decide_status( A1, A2, ... An, A1', A2', ... An');
         if flag == 0
            f :: T has no relation with f' :: T'
         if flag == 1
            f :: T = f' :: T'
         if flag == 2
            f :: T > f' :: T'
            if flag == 3
                $f :: T < f' :: T'$
    else if $B < B'$
        find type variable $Tv$ in $B'$ and corresconding more specific type $Ts$ in $B$
        replace every $Tv$ in $B'$ with $Ts$, get new argument types $A1\_new'$, $A2\_new'$, ..., $An\_new'$
        flag = decide_status($A1, A2, ..., An, A1\_new', A2\_new', ..., An\_new'$)
        if flag == 1
            $f :: T < f' :: T'$
        else
            $f :: T$ has no relation with $f' :: T'$
    else if $B > B'$
        find type variable $Tv$ in $B$ and corresconding more specific type $Ts$ in $B'$
        replace every $Tv$ in $B$ with $Ts$, get new argument types $A1\_new$, $A2\_new$, ..., $An\_new$
        flag = decide_status($A1\_new, A2\_new, ...An\_new, A1', A2',... An'$)
        if flag == 1
            $f :: T < f' :: T'$
        else
            $f :: T$ has no relation with $f' :: T'$
```

Here is the description of algorithm decide_status:

```
decide_status( $A1, A2, ... An, A1', A2', ... An'$)
        for $1 \le i \le n$, $1 \le j \le n$
            compare $Ai$ with $Aj'$
            if $Ai$ has no relation with all $Aj'$
                $f :: T$ has no relation with $f' :: T'$
            else
                if $Ai > Aj'$
                    flag = 2
                if $Ai = Aj'$
        flag = 1
    status_array[ i ] = flag;  // store the comparison result of two types
    if  (number of 1) == n          // check how many 1's in status_array
        return 1;
    else if  (number of 1) + (number of 2) == n
        return 2;
    else if  (number of 1) + (number of 3) == n
        return 3;
    else
        return 0;
```

We describe carefully the detailed algorithm of insertion.

1. Before any insertion can take place, the root of each file must be set. The root is a record containing the most general type of all record's type in that particular file. Furthermore they are linked together based upon the extra argument relations.

Given the user query (number_of_argument, query_function_type, function_name), decide which file will be used to insert the function, that means, which sub-library will take in the query function.

```
if number_of_argument = 1
        file:  Arg1.dat      root  record:    {a|b}{ }{ }{ }{0}
if number_of_argument = 2
        file:  Arg2.dat      root record:    {a|b|c}{ }{ }{ }{0}
if number_of_argument = 3
        file:  Arg3.dat      root record:    {a|b|c|d}{ }{ }{ }{0}
```

2. Insertion is the most complex operation in the library system. Before and after insert a new node, a set of conditions must be checked to make sure that every kind of link is correct and complete.

Compare the query_function_type ***Tq*** with the particular sub-library function type ***Tl*** starting from the root record of the sub-library. Before we insert new node into the library, we set new_node_exist = 0;

```
if   check_function(Tq, Tl) == 1                // Tl = Tq
          if  function_name field of the record is empty
```

```
                insert new_node into file
                insert source code into sublibrary
                new_node_exist = 1
        else
                if query_function_name already exists in  function_name field of record
                        message ("already exist")
                    stop
                else
                        insert new_node into file
                        insert source code into library
                        new_node_exist = 1
else if   check_function(Tq, Tl) == 2            // Tl > Tq
        if specific_link field of record is empty
                if new_node_exist = 1
                        update the existing links
                else
                        insert new_node into file
                        insert source code into sub-library
                        new_node_exist = 1
                        if   number_of argument > 1
                          call freeze
        else
                compare Tq with all the specific_linked children types Tli
                if     all the Tli have no relation with  Tq
                        insert a new_node into file
                        insert source code into sub-library
                        new_node_exist = 1
                        if   number_of argument > 1
                        call freeze
                        call checkchild
                else
                    i = 0
                  while   i < number_of_specific_links
                    compare the relation of Tq with Tli      //call  function modify(Tli, Tq)
                      if Tq has no relation with Tli
                          call checkchild
                          else
                              call insertion(Tli, Tq)          //recursion
else if  check_unction(Tq, Tl) == 3       //Tl < Tq
    if  new_node_exist = 0
            insert new_node into file
            insert source code into sub-library
            if  number_of_argument > 1
                  call freeze
    else update links
```

3.2.2 Retrieval

Retrieval can be seen as the major purpose of **WISER** system. The system allows for retrieving exact matched functions as well as general matched, specific matched and extra-argument matched functions. Retrieval conforms to the process of navigating the graph constructed by the various links among components in the library. Figure 19 shows the structure of the retrieval process.

According to Figure 19, the user submits his query in the form of the function's type and the number of arguments contained in it. The argument types can be given in any order. A confirmation is then asked by the system so that changes can be made if the query contains some errors. Next a search is done on the library. The search is first narrowed down by focusing only on those components containing the number of arguments specified by the user. This is implemented by using the file containing the number of arguments specified.

If a query type is an instance of a type that belongs to a record in the library, then you need only check the descendents of that record. Table 5 outlines function **exact_match** that can be used to find functions corresponds to the query type. Before calling this function, a check is made with the root record of the corresponding argument file to determine if an exact match of type already exists, in which case it would return the functions associated with that record and also its offset. Otherwise, **exact_match** is called with its parameters being instantiated to the query type, zero (0) and the number of arguments contained in the type, respectively. By using the record stored at record_offset, the function can go through all that record's children until one is found that either performs an exact match, in which case the components stored with it are returned along with its offset, or is more general than the query type, in which case the search is restricted to the descendents of it. Furthermore this function calls itself recursively thereby narrowing down the search further.

During the retrieval process, if a record is found whose **set_type** produces an exact match with the query type, the functions associated with that record will be retrieved, the offset of the record is also remembered. Furthermore, if the exact match result can not satisfy the user's requirement, the relaxed matches can be achieved by traversing the library using the various linked offsets associated with this record. If an exact match can not be found, the retrieval process is terminated. Exact match is the basis for all the relaxed matches.

Figure 19 Structure of Retrieval

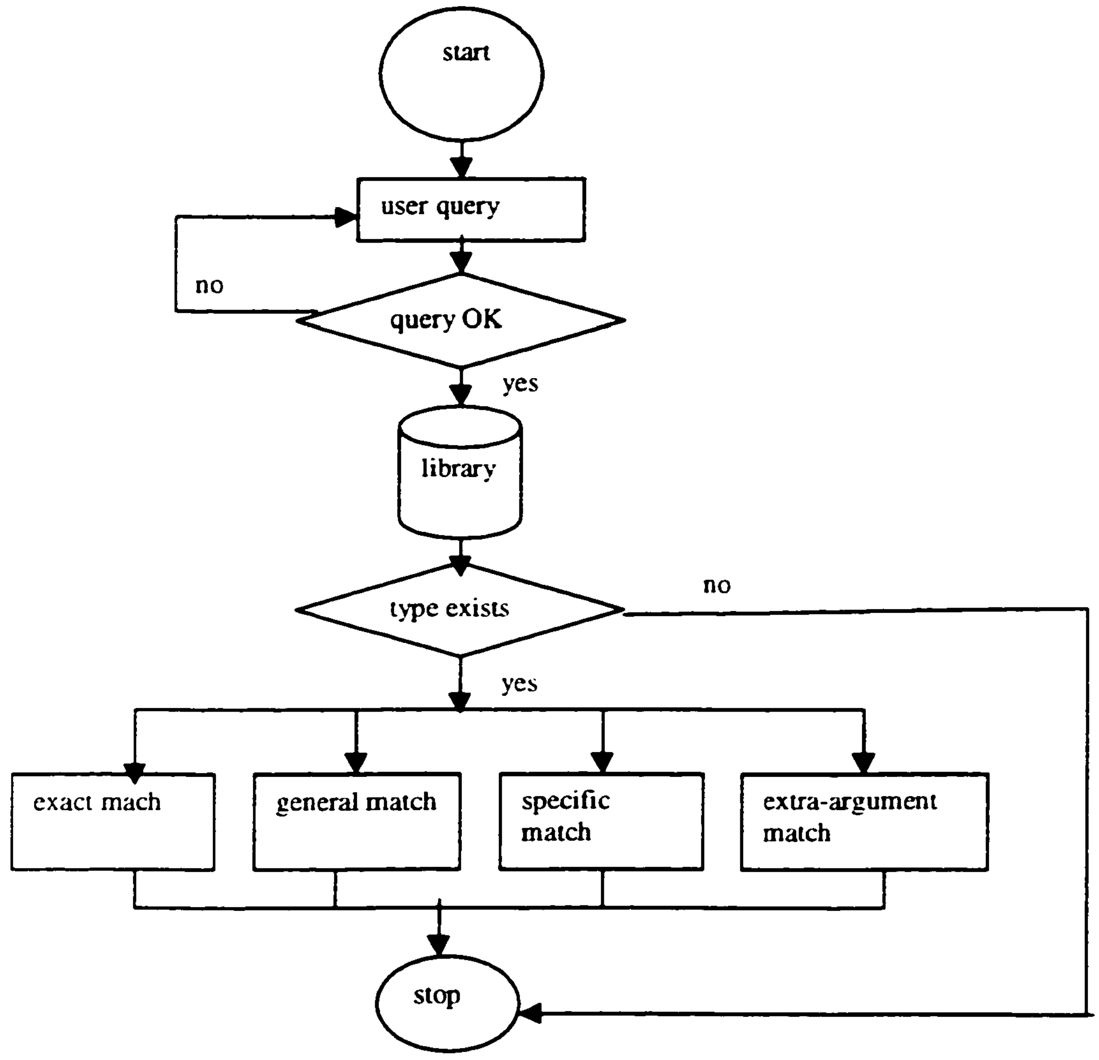


Table 7 Exact_match algorithm

```
Function exact_match(query_type, record_offset, num_args)
  get all children C1, C2 ...Cn of record at record_offset;
  for I = C1 ...Cn
```

```
    get type Ti from Ci;
    type_offset = offset of Ci;
    if retrieval_type < Ti then
      if retrieval_type == Ti
        return function names along with type_offset;
        found = true;
        exit for loop;
     end if
    end for loop
    if found == true
      found = false;
      return exact_match(retrieval_type, type_offset, num_args);
    end if
end Function
```

There are two ways in which the method **exact_match** determines when the query type does not match the library type. The first possibility is that we meet a leaf record that is more general than the **query_function_type** (See the following example 2). The second possibility is that we are at a record whose type is more general than the **query_function_type** but none of its children performs an exact type match and also none is more general than the query type (See example 4).

The **exact_match** method returns the offset of the record which contains the type that performed the exact match, because this offset will be used to find general, specific and extra-argument matches.

The process of finding more general functions is accomplished by returning all functions stored with records that are ancestors of the record. This is done recursively by using the general offset links. The same is done for more specific functions, the only difference is that it uses the descendants of the record along with the specific offset links. Finally, for the extra argument functions, the file containing types with one more argument than the **query_function_type** is used and the functions associated with the offsets stored in the record's extra_argument offset link field are retrieved. Taking **general_match** as example, let's describe the algorithm which is similar as **specific_match** and **extra_argument_match**:

```
get exact_match_offset and seek the record in this offset
find all the general_match_offset for  this record
if general_match_offset field is empty
        "no general match functions"
         stop
else
        go to all the record whose offset equals general_match_offset and
         retrieve all the function names from function_name field
        return the function_names
```

By studying the retrieval process, we have the following observations:

1. Retrieval is firstly narrowed down in terms of the number of arguments.
2. Retrieval process is applied recursively.
3. Exact match is the basis to other relaxed matches.

Figure 20 gives some examples of retrieval paths. From the graph, we can see that there exist more than one search path leading to a component. Such as ***num -> num***. You can follow path ***a -> b*** to ***a -> num*** to ***num -> num***, and you can also follow path ***a -> b*** to ***a -> a*** to ***num -> num***. But in our algorithm, there is no ambiguity. Once we find the relation between the query type and library component type, we do not need to check its sibling(s) of the library node. The order of the nodes at the same level (all the children of a parent) is decided at insertion time. For intuition, the same level nodes in this graph are listed from left to right according to their insertion order. For example, ***a -> num*** is inserted before ***a -> a***. ***{a, b} -> a*** is inserted before ***{(a -> a -> a), [a]} -> a***. This convention in this graph aids us to understand in the following examples why we do not need to investigate all the components in the library for a match.

Example 1: This example is to search function(s) with type ***[a] -> [a]***, we have the following comparisons:

[a] -> [a] < ***a -> b***

[a] -> [a] has no relation with ***a -> num***

[a] -> [a] < ***a -> a***

[a] -> [a] has no relation with ***num -> num***

[a] -> [a] = ***[a] -> [a]***

So we find the exact match. Based on the exact match, we find its general match type is ***a-> a***, extra-argument match type is ***{num, [a]} -> [a]***.

Example 2: The second example uses ***[num] -> [num]*** as search key. We have the following comparisons:

[num] -> [num] < ***a***

[num] -> [num] < ***a -> num***

[num] -> [num] has no relation with ***num -> num***

[num] -> [num] < ***a -> a***

[num] -> [num] has no relation with ***num -> num***

[num] -> [num] < ***[a] -> [a]***

We already reach the leaf node ***[a] -> [a]*** in the graph and still can not find the exact match type. This implies that the query type is not in the library. The query ***[num] ->[num]*** fails yielding exact match.

Example 3: The query type is ***{[char], num}-> [char]***. We have the following comparisons during the search:

{[char], num}-> [char] < ***{a, b} -> c***

{[char], num}-> [char] < ***{a, b} -> a***

[char], num}-> [char] has no relation with ***{a, a} -> a***

{[char], num}-> [char] < ***{num, [a]} -> [a]***

{[char], num}-> [char] = ***{num, [char]} -> [char]***

Finally we find the exact match node. Based on this exact match, we find the general match type is ***{num, [a]} -> [a]***.

Example 4: The example deals with query type ***{[num], num} -> [num]***. We know:

{[num], num} -> [num] < ***{a, b} -> c***

{[num], num} -> [num] < ***{a, b} -> a***

{[num], num} -> [num] has no relation with ***{a, a} -> a***

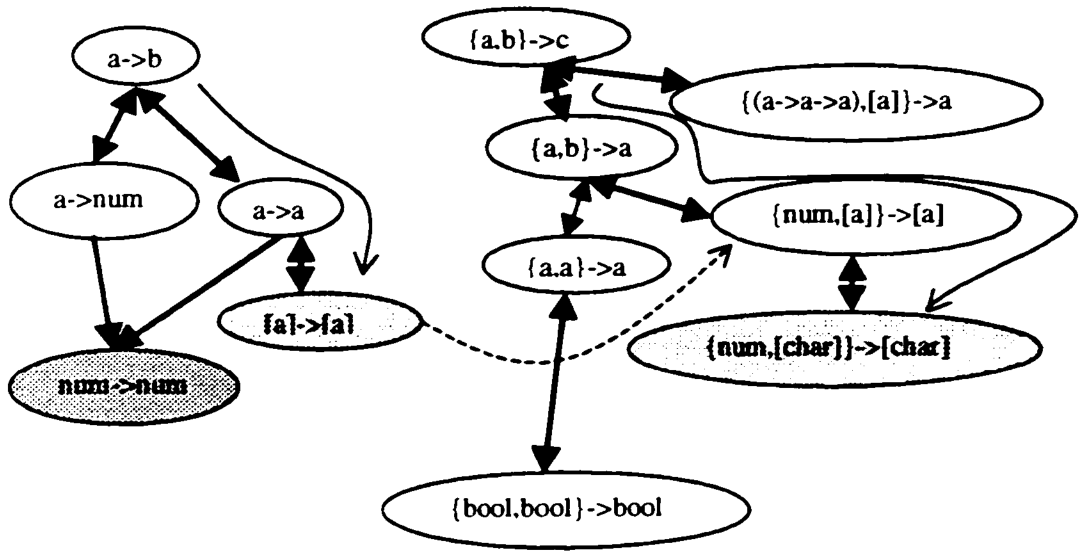
{[num], num} -> [num] < ***{num, [a]} -> [a]***

{[num], num} -> [num] has no relation with ***{num, [char]} -> [char]***

In this case, ***{num, [a]}->[a]*** is more general than our query type, but its child has no relation with our query type. We can conclude that this query type is not in the library.

From this graph, we can see that the retrieval is narrowed down by a breath-first search. The search time is greatly reduced comparing to sequential search.

Figure 20 Examples of search path



3.2.3 Deletion

A function can be implemented in many ways. If the software developer has a better implementation of a function, he may want to replace it with the better one. The replace procedure goes through two steps. Firstly, the old function needs to be deleted. Secondly, the new function needs to be inserted. It is quite possible that the new function has the same type as the old one. So when we do deletion, we just remove the function name from the corresponding record, if there is no more function names in the record after this deletion, we still keep this record, because later on an insertion on the same **set_type** will be performed. Finally, the source code of the old function is taken away from the library. This design idea is practical and makes deletion much easier. Figure 21 is the structure of deletion.

Deletion algorithm is similar to retrieval algorithm. For deletion, after you find the component with the same type and name, you just need to delete the name from the record and remove the source code from the library.

Figure 21 Structure of Deletion

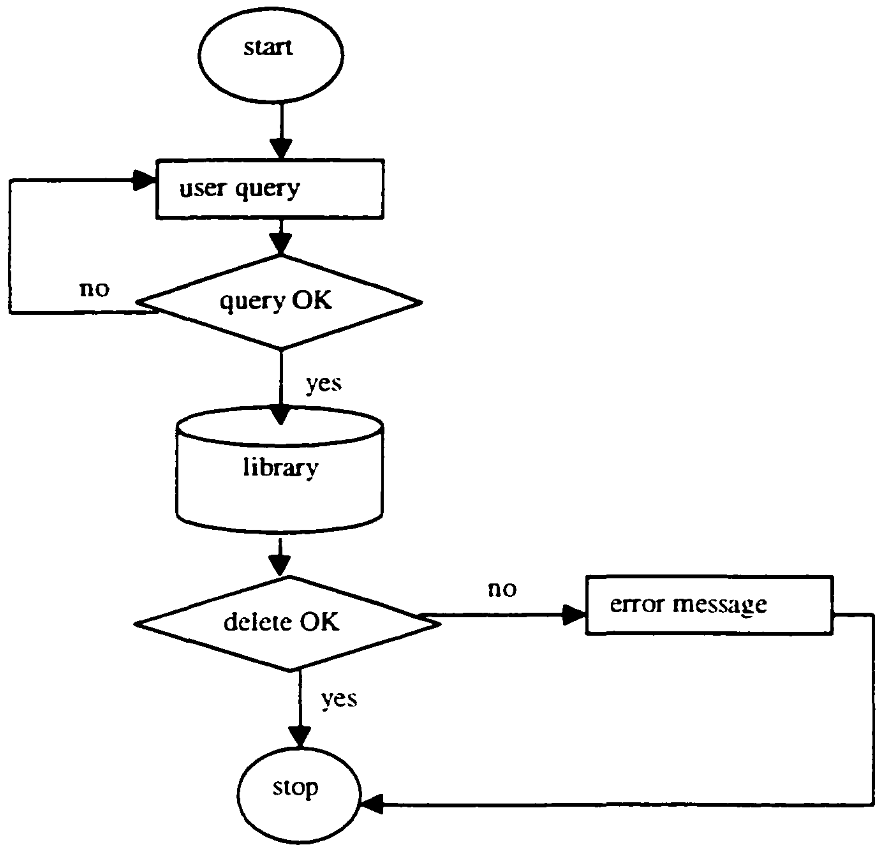


Figure 22 Example of Deletion

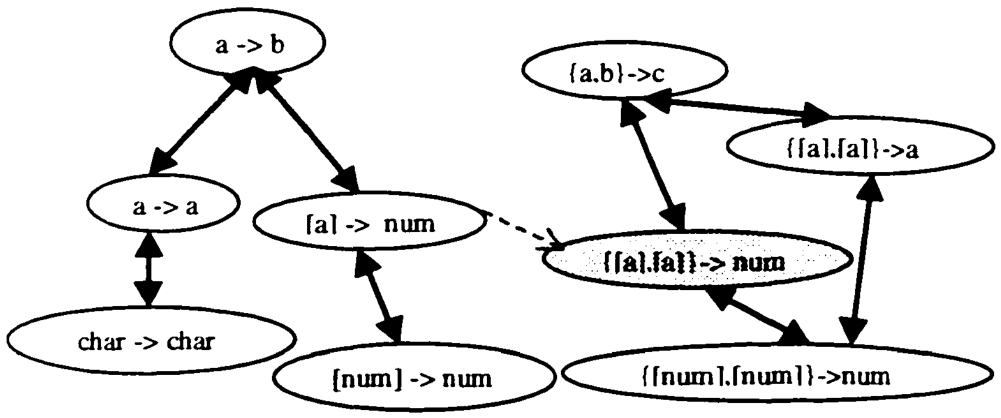
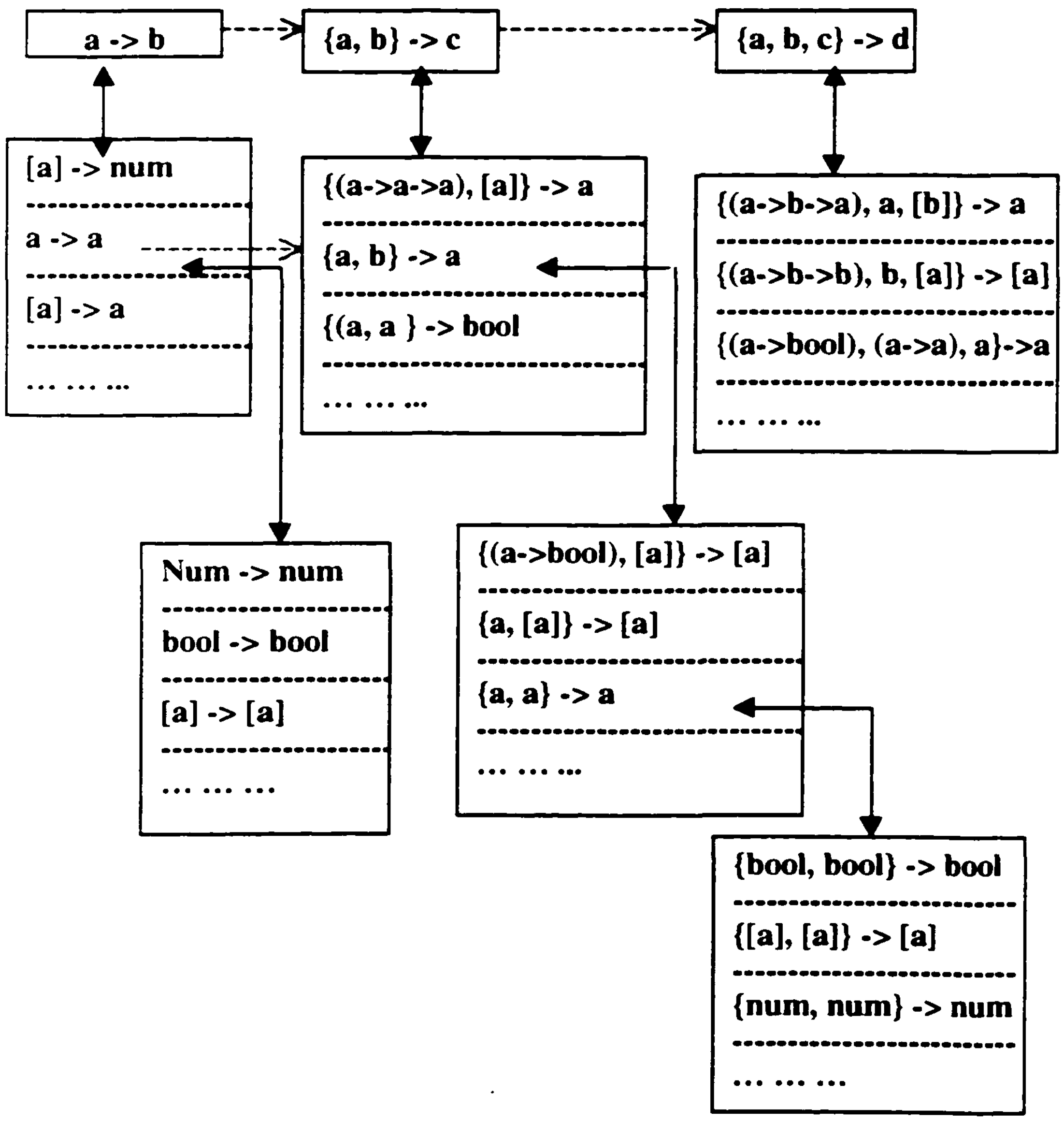


Figure 22 gives us an example of deletion. Assuming we have two functions **f1** and **f2** attached to type ***{[a], [a]} -> num***. After deleting function **f1**, the structure is the same. Now we continue to delete function **f2** from the structure. According to our design algorithm, the structure keeps the same. Now the type ***{[a], [a]} -> num*** contains zero function. In other words, we allow empty node (no function attached to the node) exist in the structured library.

3.2.4 Browsing

Figure 23 Example of browsing



Browsing is an operation that allows the library user to view the structure of the library so as to find potential reusable functions. Figure 23 shows a sample structure of browsing. Using the browser, the user can control which part of the structure to browse by changing the number of arguments along with moving up, down and cross the structure. Browsing always begins from the root node. The user can visit all the nodes in the structure based on the various links. The user can choose different paths to search for an appropriate function by using the browser.

Chapter 4 Program Development Using WISER

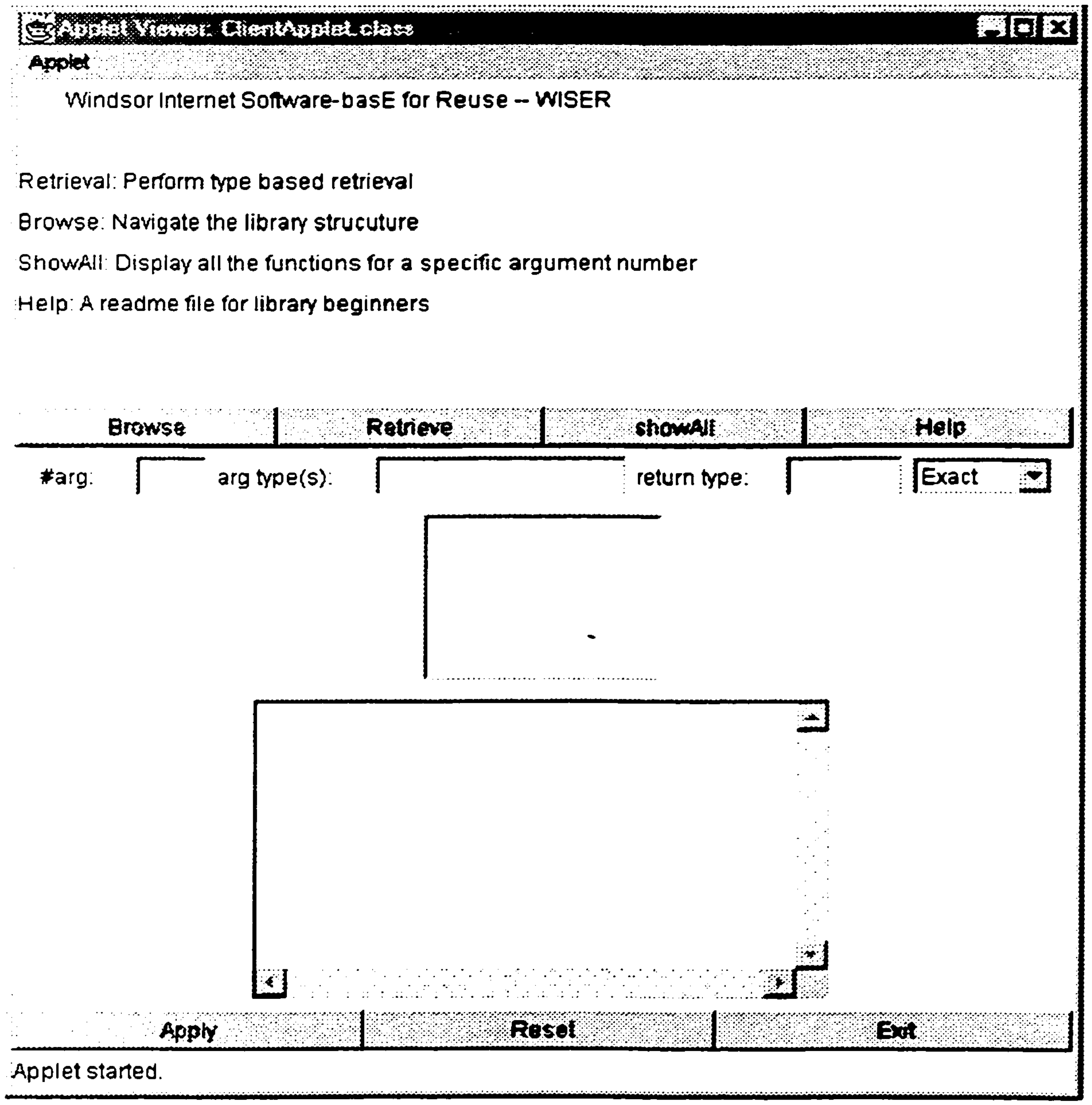
This chapter describes **WISER**'s user-friendly interfaces developed by using Java programming language. Some examples will be given on the various operations applied to the system.

4.1 User Interface

On the user side, an applet window is used as the interface to get the user's query and display the feedback from the server side. Figure 24 is the user interface. The upper portion of the user interface is a brief description of various operations supported by WISER. They include *browse*, *retrieve*, *showAll* and *help*. The lower portion of the user interface composes three panels. The north panel contains four buttons listed sequentially from left to right, named "**Browse**", "**Retrieve**", "**showAll**" and "**Help**". The user chooses one operation by clicking on one of the buttons.

The center panel is further divided into three parts. The upper part is used to collect user's input. In this part, there are three *textFields* and one *choiceButton*. One *textField* is the ***#arg*** *textField*, another is ***arg type(s)*** textField. The *choiceButton* is provided with four kinds of matches, "***Exact***", "***General***", "***Specific***" and "***ExtraArg***". The middle part of the center panel is a list for displaying the array of candidate function names. The lower part of the center panel is a *textArea*. When the user click one of the items (function names) in the list of middle part, the source code of this particular function is displayed in this *textArea*.

Figure 24 User Interface



Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval

Browse: Navigate the library strucuture

ShowAll: Display all the functions for a specific argument number

Help: A readme file for library beginners

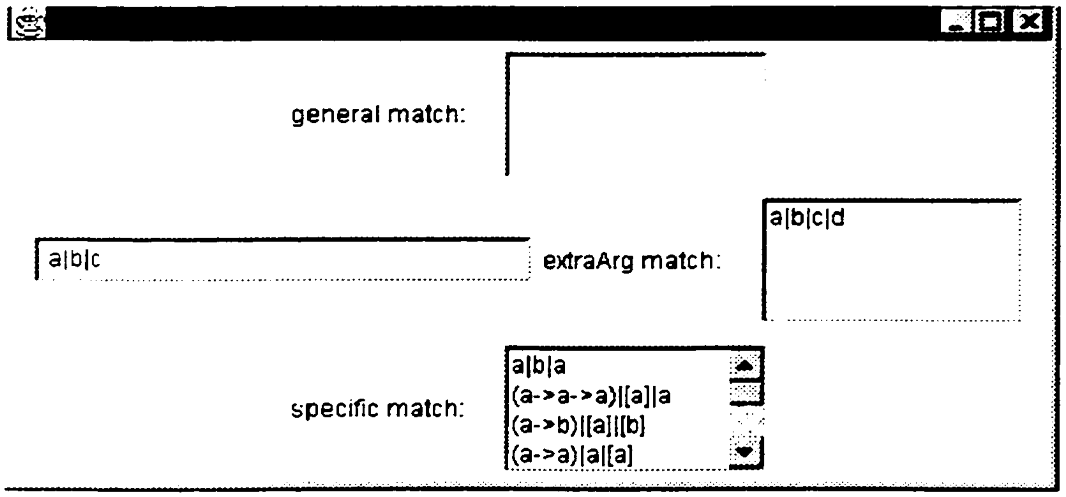
Applet started.

The south panel contains three buttons. When "**Apply**" is clicked, the user's query will be sent to server through the socket connection. When "**Reset**" is clicked, the user's previous input shown on the window will be cleared. The "**Reset**" button is used to start a new transaction. "**Exit**" is used to terminate this user's session. All the interactions with WISER are processed through this user interface.

4.1.1 Example of Browser

A browser is implemented to allow the user to view the software reuse library structure and find potential candidate functions. Figure 25 is the browser window. It pops up when you clicks the "**Browse**" button and specify the **#arg** on the User Interface Main Window. The browsing procedure always begins from the root of corresponding sub-library.

Figure 25 Browser Window for #arg = 2



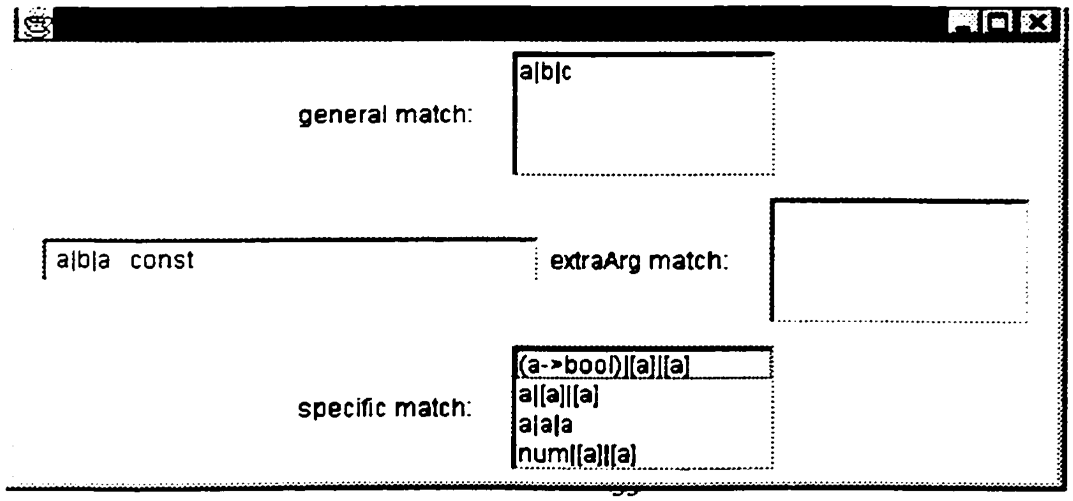
general match:

a|b|c

extraArg match: a|b|c|d

specific match:
a|b|a
(a->a->a)|[a]|a
(a->b)|[a]|[b]
(a->a)|a|[a]

Figure 26 Browing ***(a, b) -> a***



general match: a|b|c

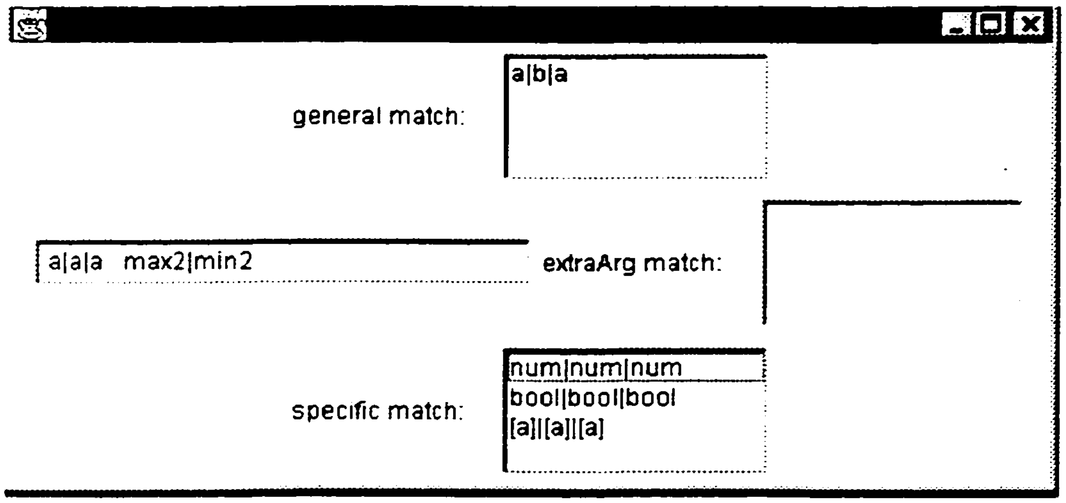
a|b|a const

extraArg match:

specific match:
(a->bool)|[a]|[a]
a|[a]|[a]
a|a|a
num|[a]|[a]

From this window you can see all the **set_types** associating with a particular **set_type** (*{a, b} -> c* in this case, represented by ***a|b|c***). By clicking one of the **set_types** in the three lists, the user can get the functions bearing this **set_type** and all the **set_types** of immediately more general, more specific and having extra argument(s). Figure 26 is an example on clicking on ***a|b|a***. Figure 27 is an example on clicking on ***a|a|a***.

Figure 27 Browsing *{a, a} -> a*



general match:

a|b|a

a|a|a max2|min2

extraArg match:

specific match:

num|num|num
bool|bool|bool
[a]|[a]|[a]

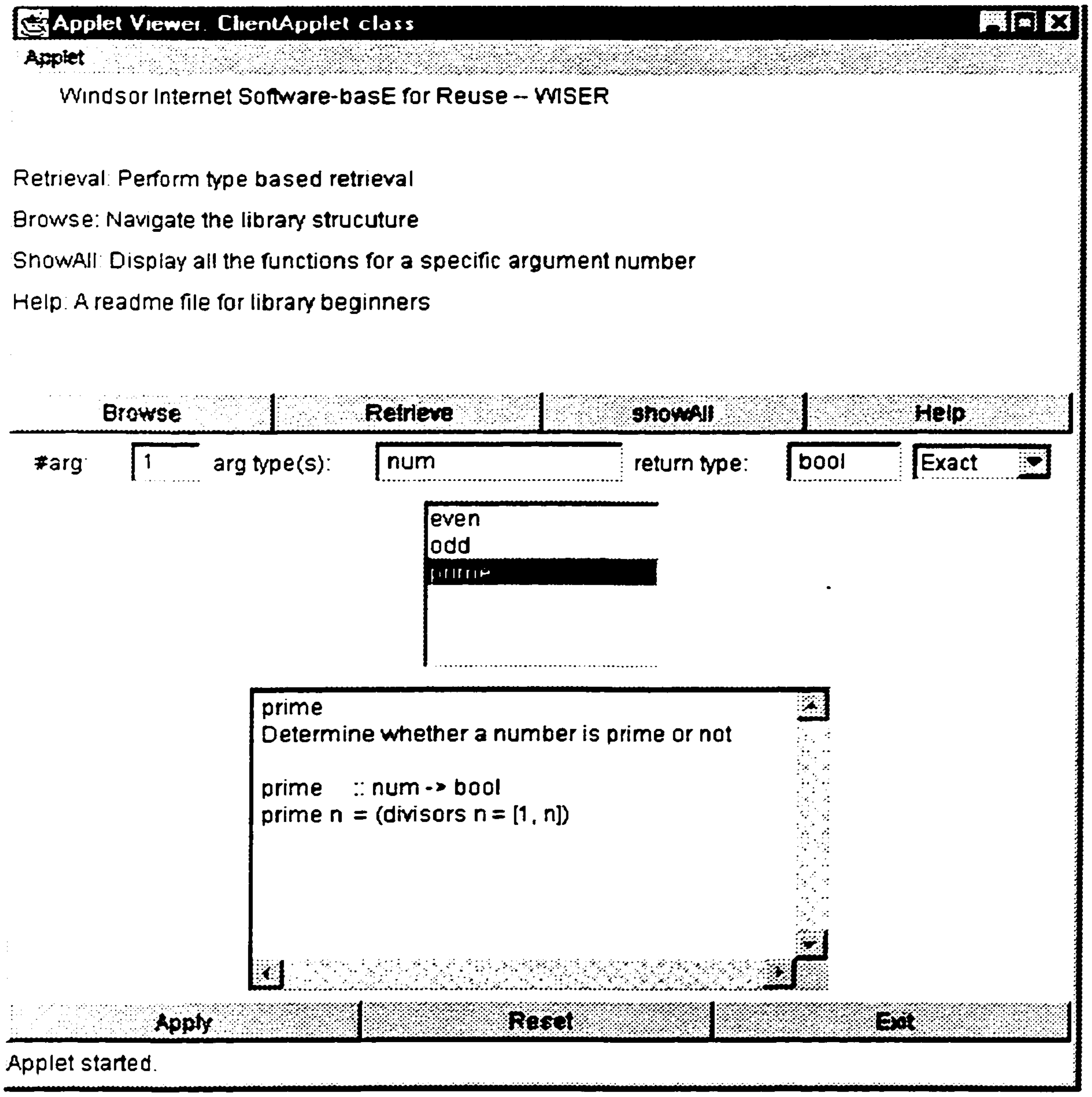
From Figure 27 we can see that there are two functions bearing **set_type *{a, a} -> a***, which are ***max2*** and ***min2***. Set type ***{a, a} -> a*** has one immediate general set type ***{a, b) -> a*** and three immediate specific set types ***{num, num} -> num***, ***{bool, bool} -> bool*** and *{[a], [a]} -> [a]*. Following the links a user can traverse the whole library in order to find the desired candidates.

4.1.2 Examples of Retrieval

4.1.2.1 Exact Match

Example 1: Suppose we want to find a function which can determine if a given number is a prime or not. We query **WISER** using type ***num -> bool***. By specifying the ***#arg = 1***, ***arg type(s) = num***, ***return type = bool*** and performing an exact match, we got three functions with type ***num -> bool***. By investigating the source code, we find function "***prime***" is our desired function. Figure 28 is the window for this retrieval.

Figure 28 Exact Match of *num->bool*



Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval
Browse: Navigate the library strucuture
ShowAll: Display all the functions for a specific argument number
Help: A readme file for library beginners

#arg: 1 arg type(s): num return type: bool Exact

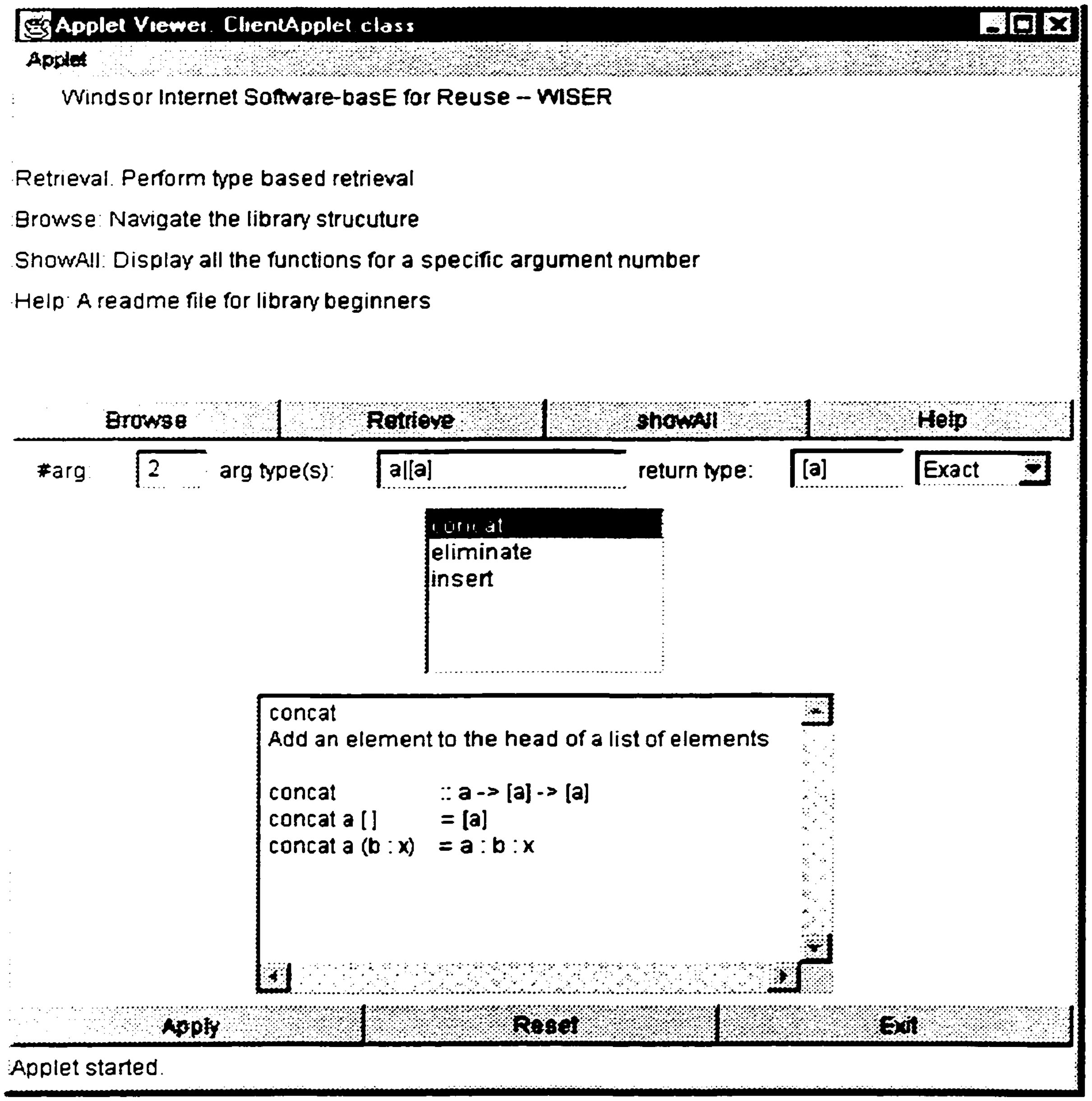
even
odd
prime

```
prime
Determine whether a number is prime or not

prime    :: num -> bool
prime n  = (divisors n = [1, n])
```

Applet started.

Example 2: Figure 29 is an example of retrieving a function which concatenates an element to the head of a list of elements. We use ***a -> [a] -> [a]*** as query type. Finally we find function "***concat***" fulfills our requirement.

Figure 29 Exact Match of *a -> [a]-> [a]*



Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval
Browse: Navigate the library strucuture
ShowAll: Display all the functions for a specific argument number
Help: A readme file for library beginners

#arg: 2 arg type(s): a|[a] return type: [a] Exact

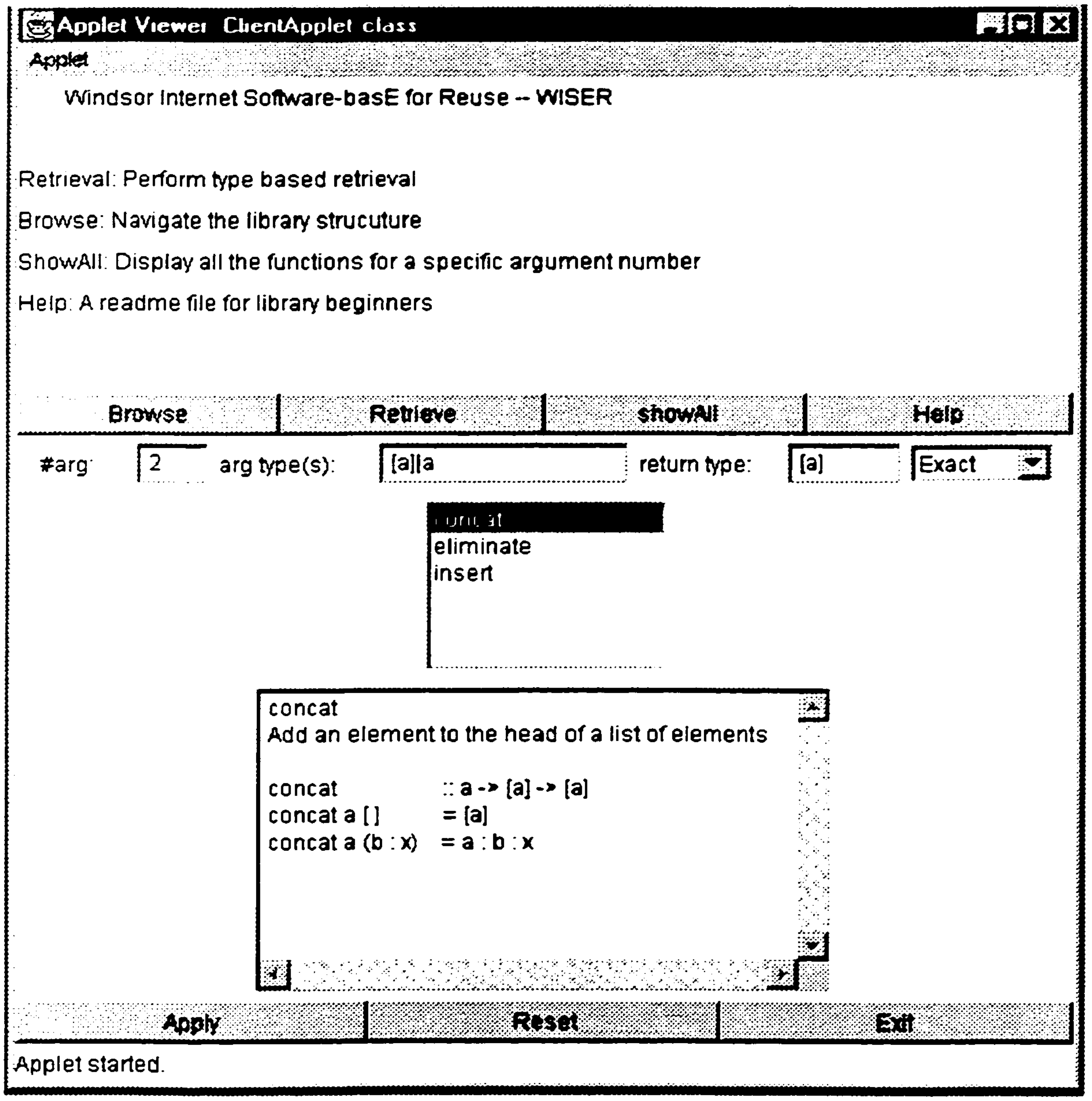
concat
eliminate
insert

```
concat
Add an element to the head of a list of elements

concat            :: a -> [a] -> [a]
concat a [ ]      = [a]
concat a (b : x)  = a : b : x
```

Applet started.

WISER is designed to disregard the difference of argument order. Figure 30 is the same retrieval as figure 29 except the argument order of query type is different. The same retrieval result has been achieved.

Figure 30 Exact Match of *[a] -> a -> [a]*



Applet Viewer ClientApplet class

Applet

Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval
Browse: Navigate the library strucuture
ShowAll: Display all the functions for a specific argument number
Help: A readme file for library beginners

#arg: 2 arg type(s): [a]|a return type: [a] Exact

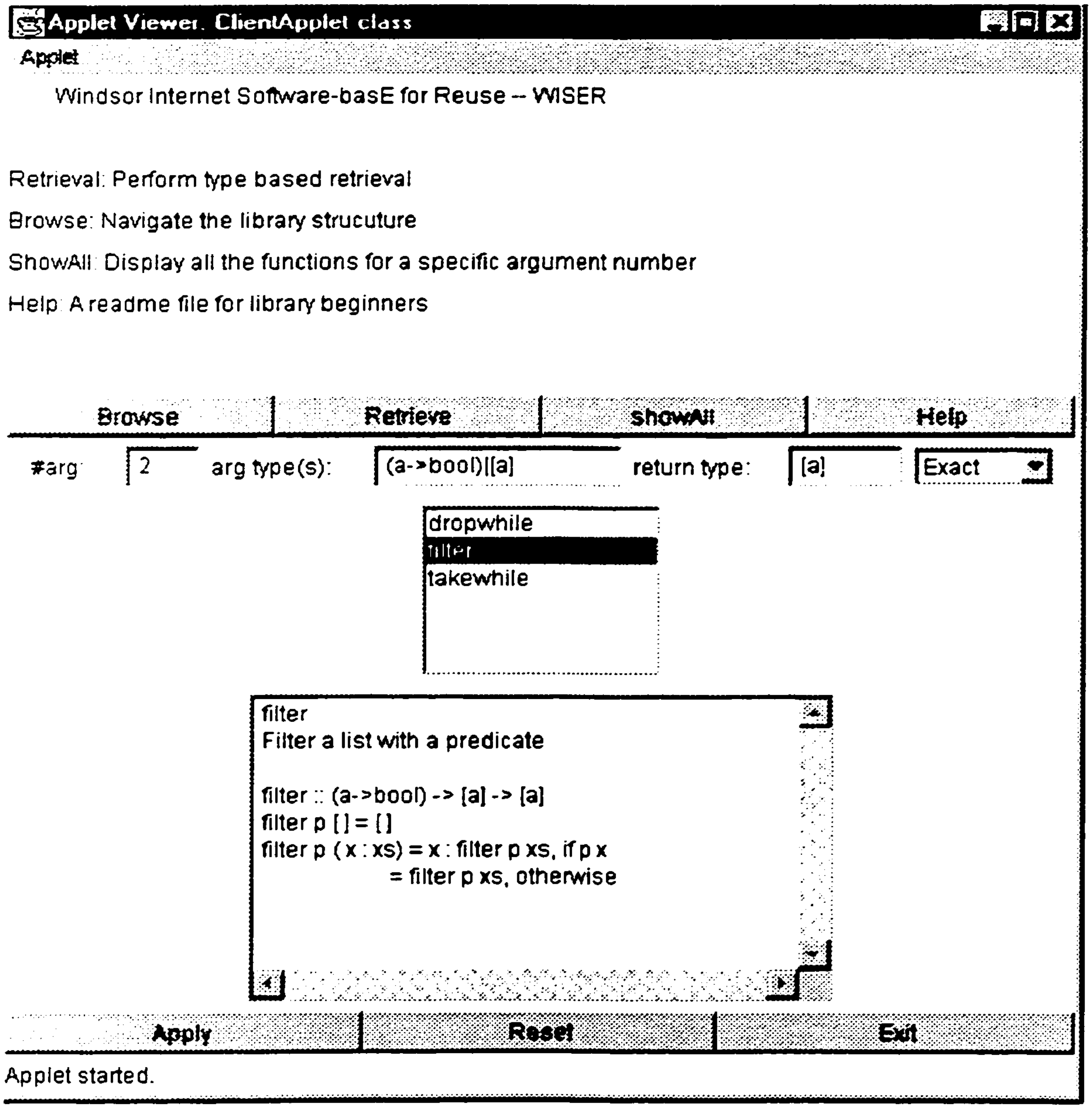
concat
eliminate
insert

```
concat
Add an element to the head of a list of elements

concat          :: a -> [a] -> [a]
concat a [ ]    = [a]
concat a (b : x)  = a : b : x
```

Applet started.

Example 3: Suppose we want to find a function which can filter a list with a predicate. We use query type ***(a -> bool) -> [a] -> [a]*** as search key and perform exact match operation. we find the desired result as shown in Figure 31. Changing the argument order of ***(a->bool)*** and ***[a]***, the query type is changed to ***[a] -> (bool) -> [a]***. We get the same retrieval result based on this query type, as shown in Figure 32.

Figure 31 Exact Match of *(a->bool) -> [a] -> [a]*



Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval

Browse: Navigate the library strucuture

ShowAll: Display all the functions for a specific argument number

Help: A readme file for library beginners

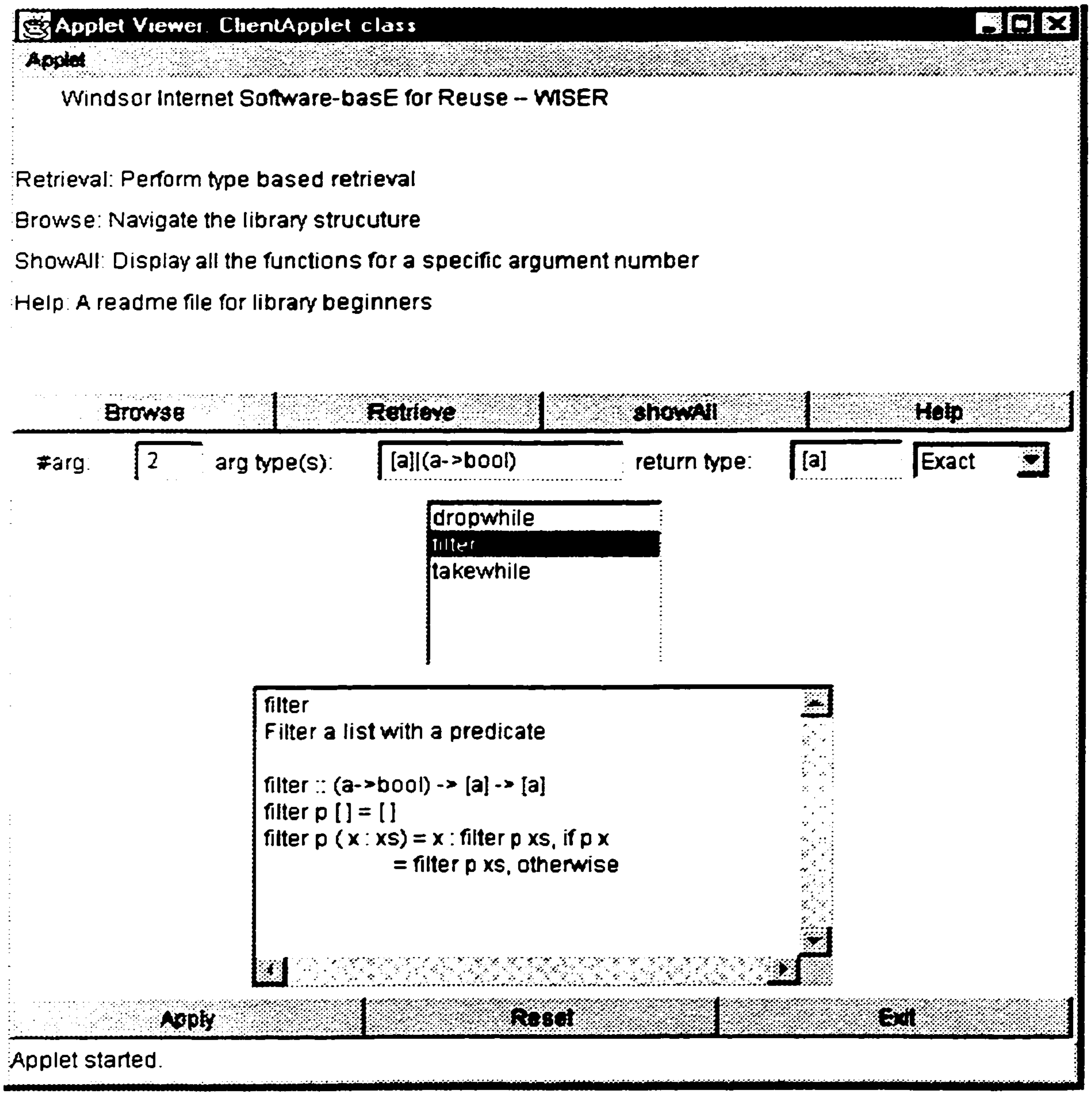
#arg: 2 arg type(s): (a->bool)|[a] return type: [a] Exact

dropwhile
filter
takewhile

```
filter
Filter a list with a predicate

filter :: (a->bool) -> [a] -> [a]
filter p [ ] = [ ]
filter p ( x : xs) = x : filter p xs, if p x
                   = filter p xs, otherwise
```

Applet started.

Figure 32 Exact Match of *[a] -> (a->bool) -> [a]*



Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval

Browse: Navigate the library strucuture

ShowAll: Display all the functions for a specific argument number

Help: A readme file for library beginners

#arg: 2 arg type(s): [a]|(a->bool) return type: [a]

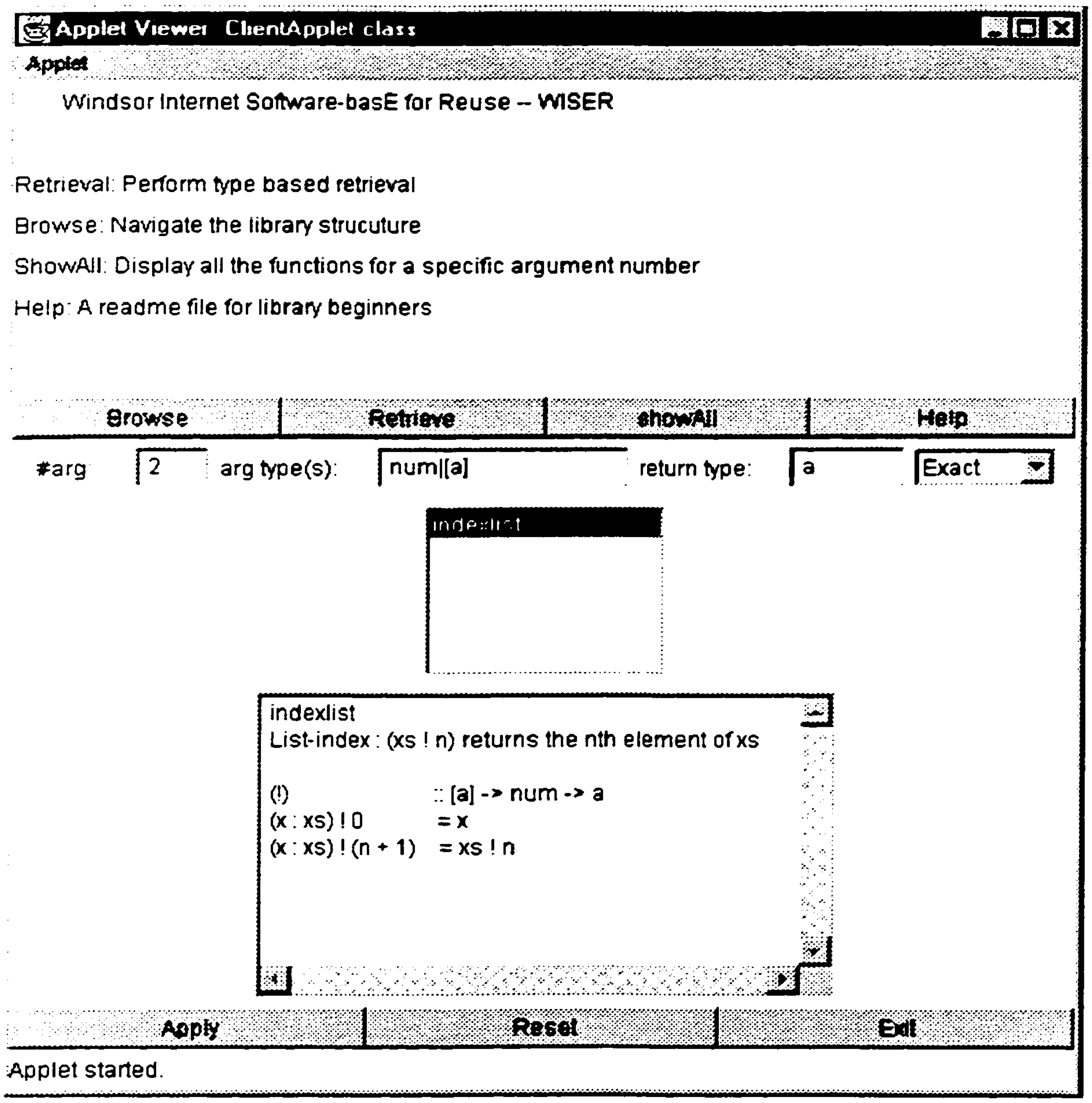
dropwhile
filter
takewhile

```
filter
Filter a list with a predicate

filter :: (a->bool) -> [a] -> [a]
filter p [ ] = [ ]
filter p ( x : xs) = x : filter p xs, if p x
                   = filter p xs, otherwise
```

Applet started.

Example 4: If we know the index of an element in a list and want to find this element from the list, we can query **WISER** with type ***num -> [a] -> a***. Figure 33 shows us the exact match result. If we reverse the order of ***[a]*** and ***num*** and use ***[a] -> num -> a*** as search key, the result is the same as shown in Figure 34.

Figure 33 Exact match of *num -> [a] -> a*


Applet Viewer ClientApplet class
Applet
Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval
Browse: Navigate the library strucuture
ShowAll: Display all the functions for a specific argument number
Help: A readme file for library beginners

Browse | Retrieve | showAll | Help

#arg: 2   arg type(s): num|[a]   return type: a   Exact

indexlist

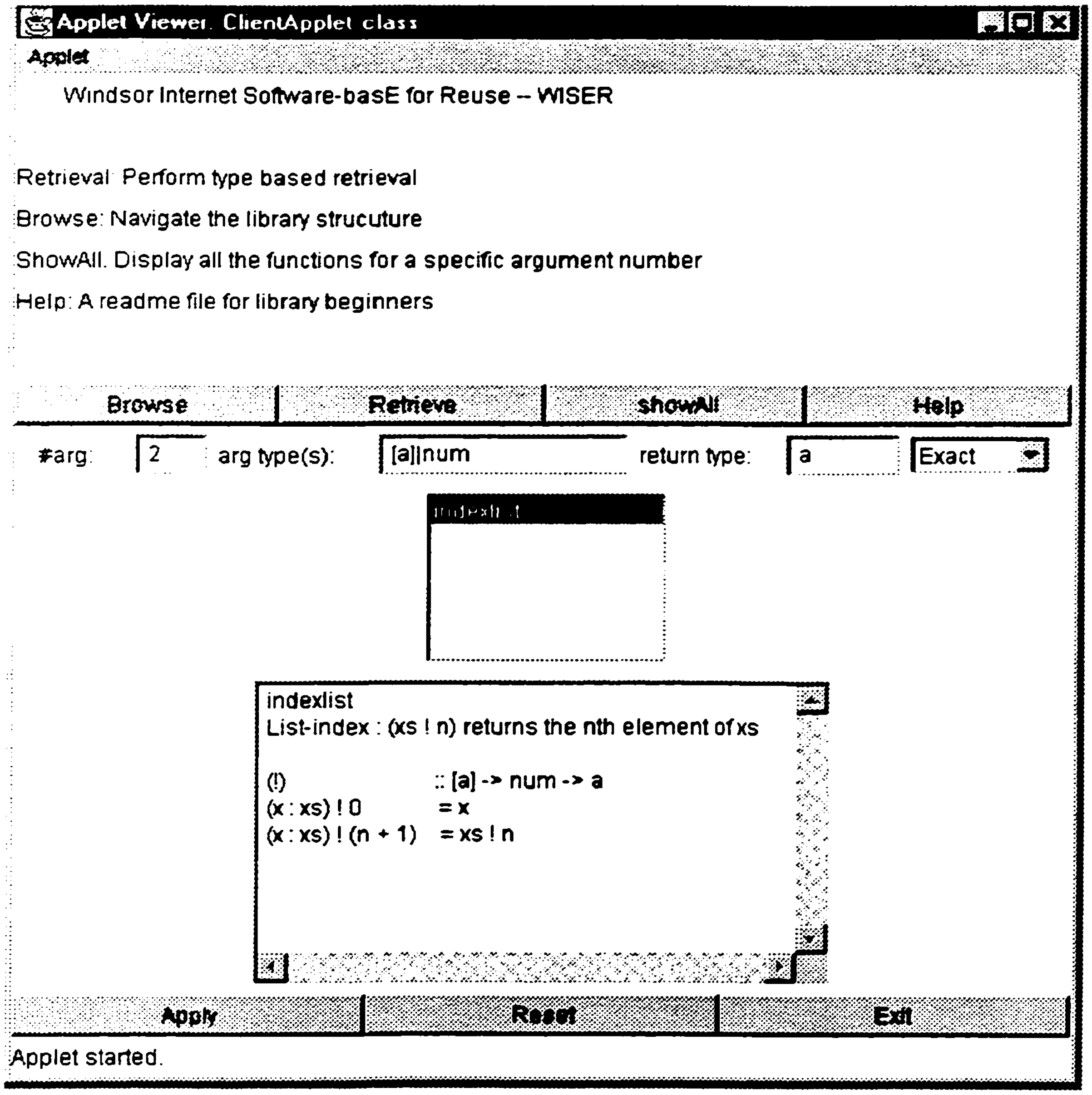
```
indexlist
List-index : (xs ! n) returns the nth element of xs

(!)               :: [a] -> num -> a
(x : xs) ! 0       = x
(x : xs) ! (n + 1) = xs ! n
```

Apply | Reset | Exit

Applet started.

Example 5: We give the last three-argument example to highlight the fact that the order of arguments can be disregarded in the retrieval by using **WISER**. Suppose we want to retrieve the source code for the important function ***foldl***, which is the building block for many useful complex functions. We use search key ***(a -> b -> a) -> a -> [b] -> a*** to query **WISER**, Figure 35 is the result. If we use ***a -> (a -> b -> a) -> [b] -> a*** as search key, Figure 36 is the result. If we use ***(a -> b -> a) -> [b] -> a*** as search key, Figure 37 is the result. There are total six permutations for three different arguments. We omit the other three permutations here.

Figure 34 Exact Match of *[a] -> num -> a*



Windsor Internet Software-basE for Reuse -- WISER

Retrieval Perform type based retrieval
Browse: Navigate the library strucuture
ShowAll. Display all the functions for a specific argument number
Help: A readme file for library beginners

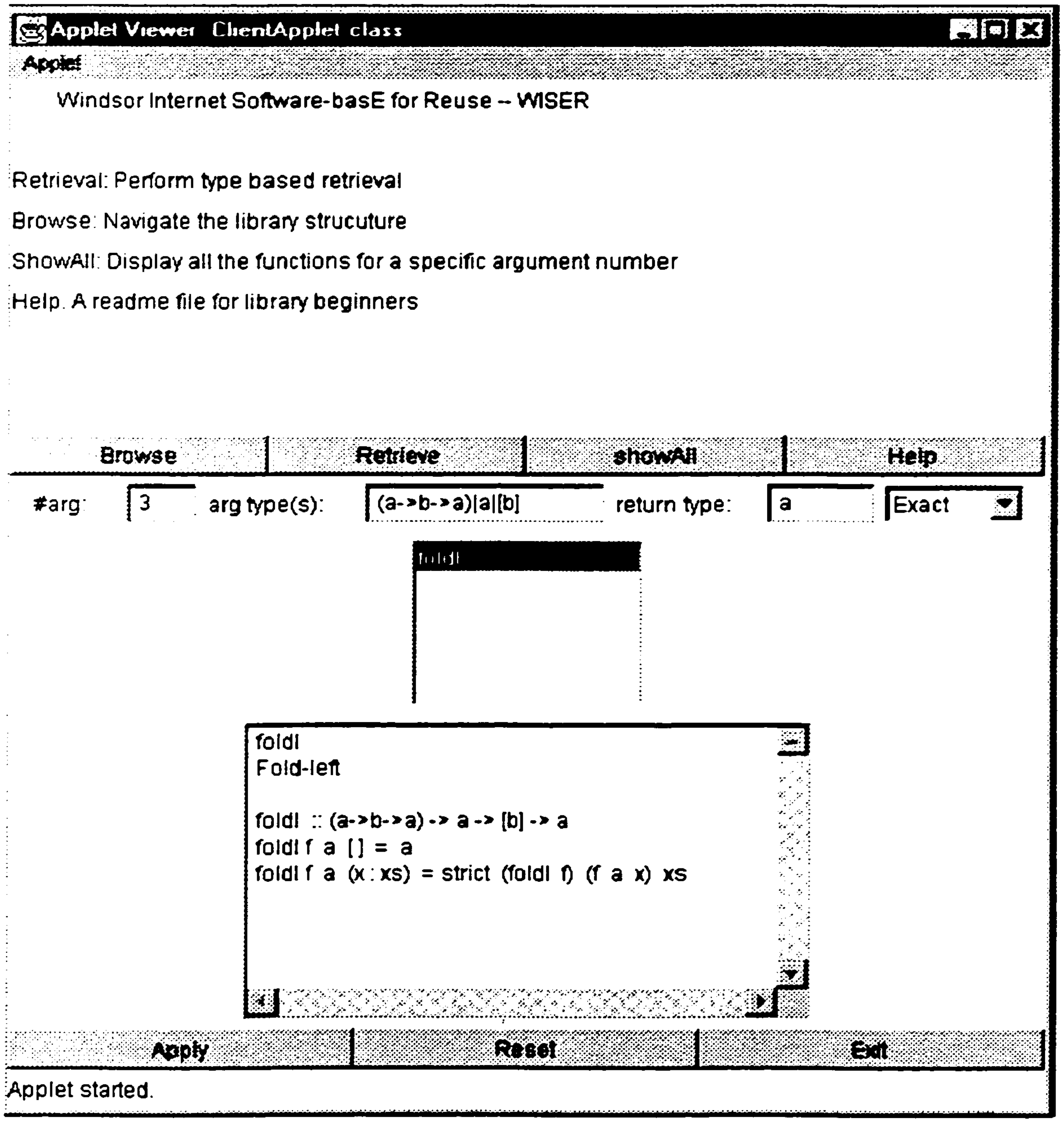
#arg: 2 arg type(s): [a]|num return type: a Exact

indexlist

```
indexlist
List-index : (xs ! n) returns the nth element of xs

(!)              :: [a] -> num -> a
(x : xs) ! 0      = x
(x : xs) ! (n + 1) = xs ! n
```

Applet started.

Figure 35 Exact Match of ***(a -> b -> c) -> a -> [b] -> a***



Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval

Browse: Navigate the library strucuture

ShowAll: Display all the functions for a specific argument number

Help. A readme file for library beginners

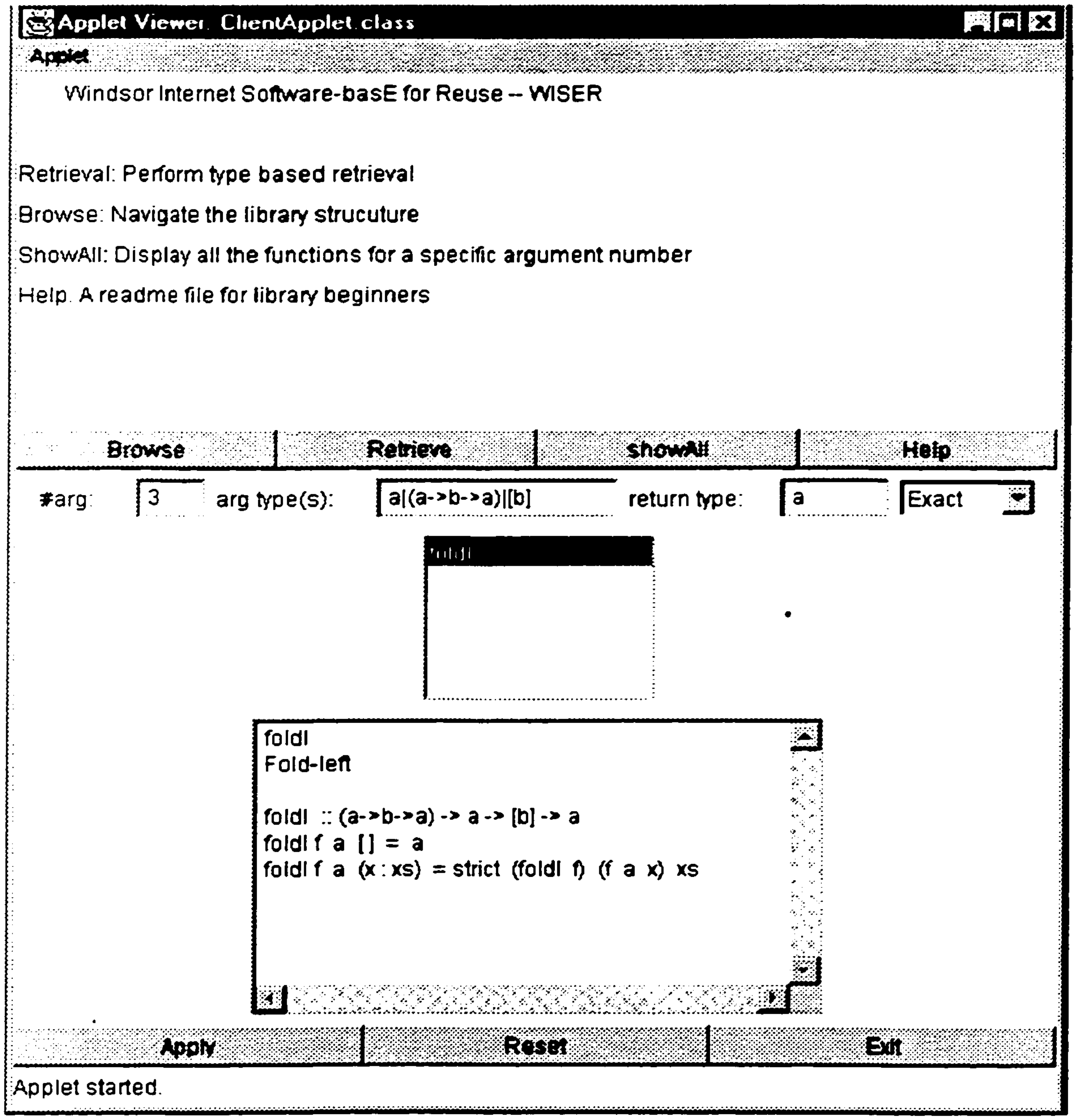
#arg: 3 arg type(s): (a->b->a)|a|[b] return type: a Exact

foldl

```
foldl
Fold-left

foldl :: (a->b->a) -> a -> [b] -> a
foldl f a [] = a
foldl f a (x:xs) = strict (foldl f) (f a x) xs
```

Applet started.

Figure 36 Exact Match of *a -> (a -> b -> a) -> [b] -> a*



Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval

Browse: Navigate the library strucuture

ShowAll: Display all the functions for a specific argument number

Help. A readme file for library beginners

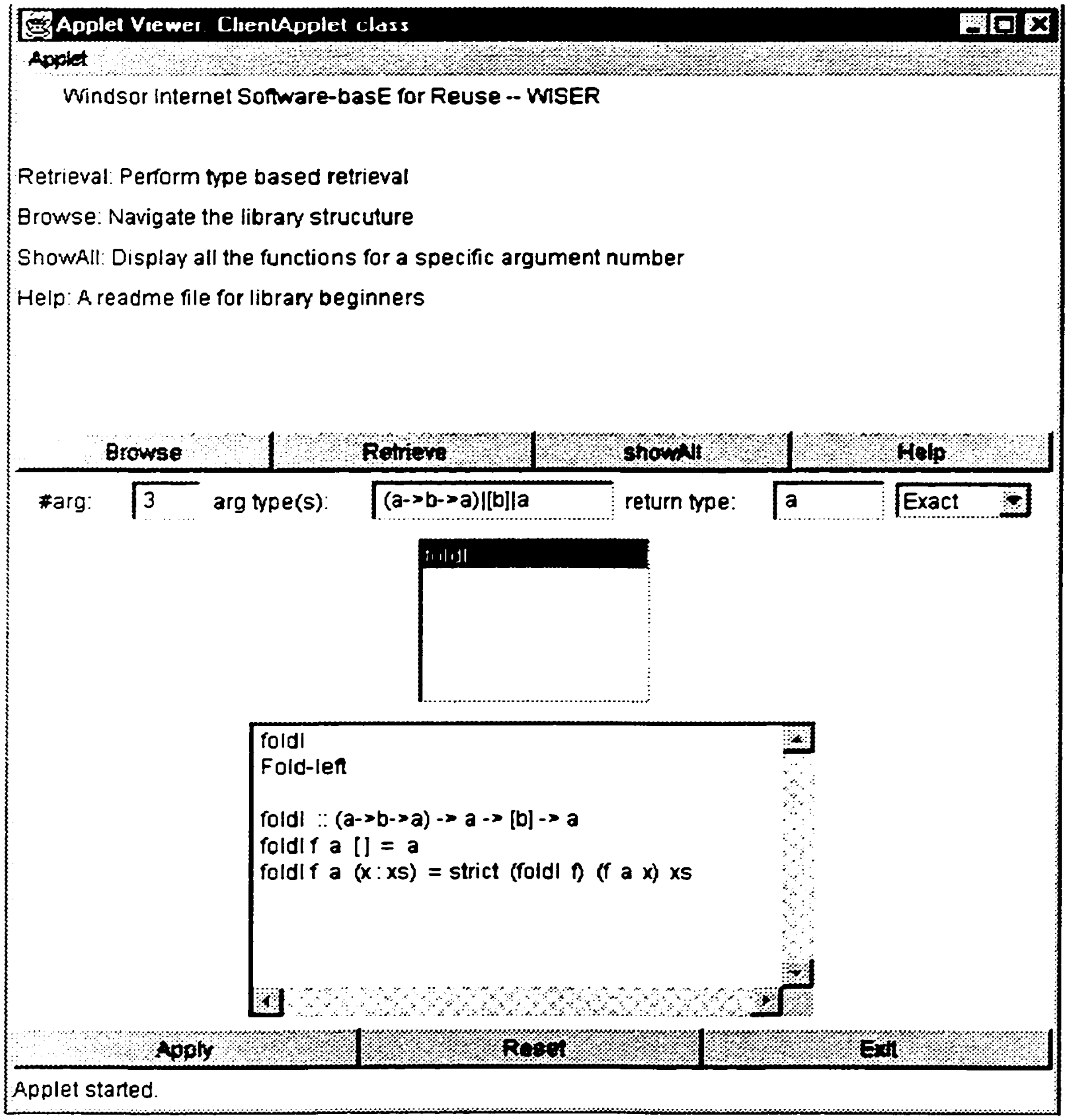
#arg: 3 arg type(s): a|(a->b->a)|[b] return type: a Exact

foldl

```
foldl
Fold-left

foldl :: (a->b->a) -> a -> [b] -> a
foldl f a [] = a
foldl f a (x : xs) = strict (foldl f) (f a x) xs
```

Applet started.

Figure 37 Exact Match of *(a -> b -> a) -> [b] -> a -> a*



Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval

Browse: Navigate the library strucuture

ShowAll: Display all the functions for a specific argument number

Help: A readme file for library beginners

#arg: 3 arg type(s): (a->b->a)|[b]|a return type: a Exact

foldl

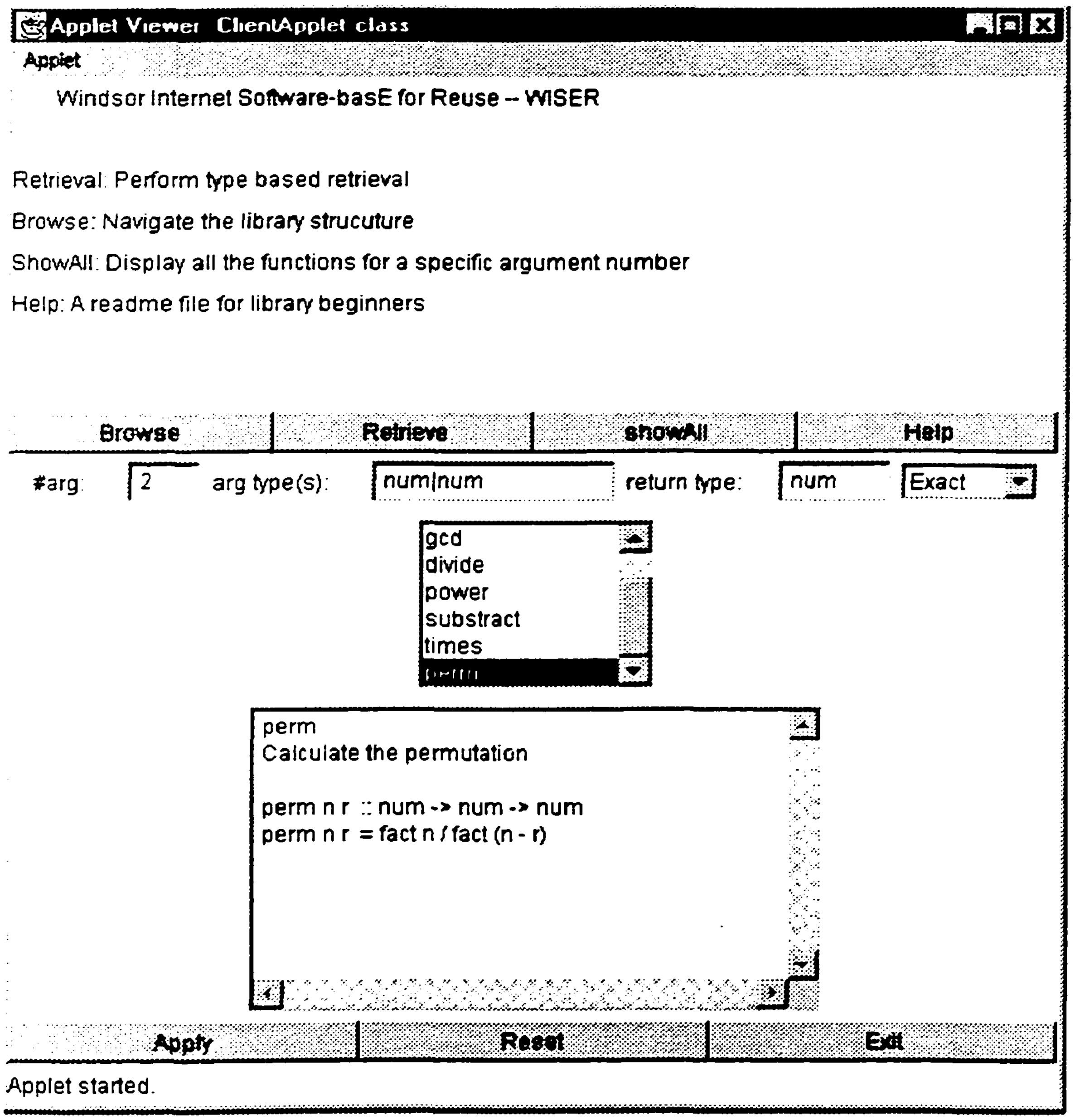
```
foldl
Fold-left

foldl :: (a->b->a) -> a -> [b] -> a
foldl f  a  [] = a
foldl f  a  (x:xs)  = strict  (foldl  f)  (f  a  x)  xs
```

Applet started.

From the above five examples we can prove that the retrieval result in **WISER** is not affected by the order of argument types. You can always retrieve the same function(s) if you use the same **set_type.**

4.1.2.2 General Match

Figure 38 Exact match of *num -> num -> num*



Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval

Browse: Navigate the library strucuture

ShowAll: Display all the functions for a specific argument number

Help: A readme file for library beginners

#arg: 2 arg type(s): num|num return type: num Exact

```
gcd
divide
power
substract
times
perm
```

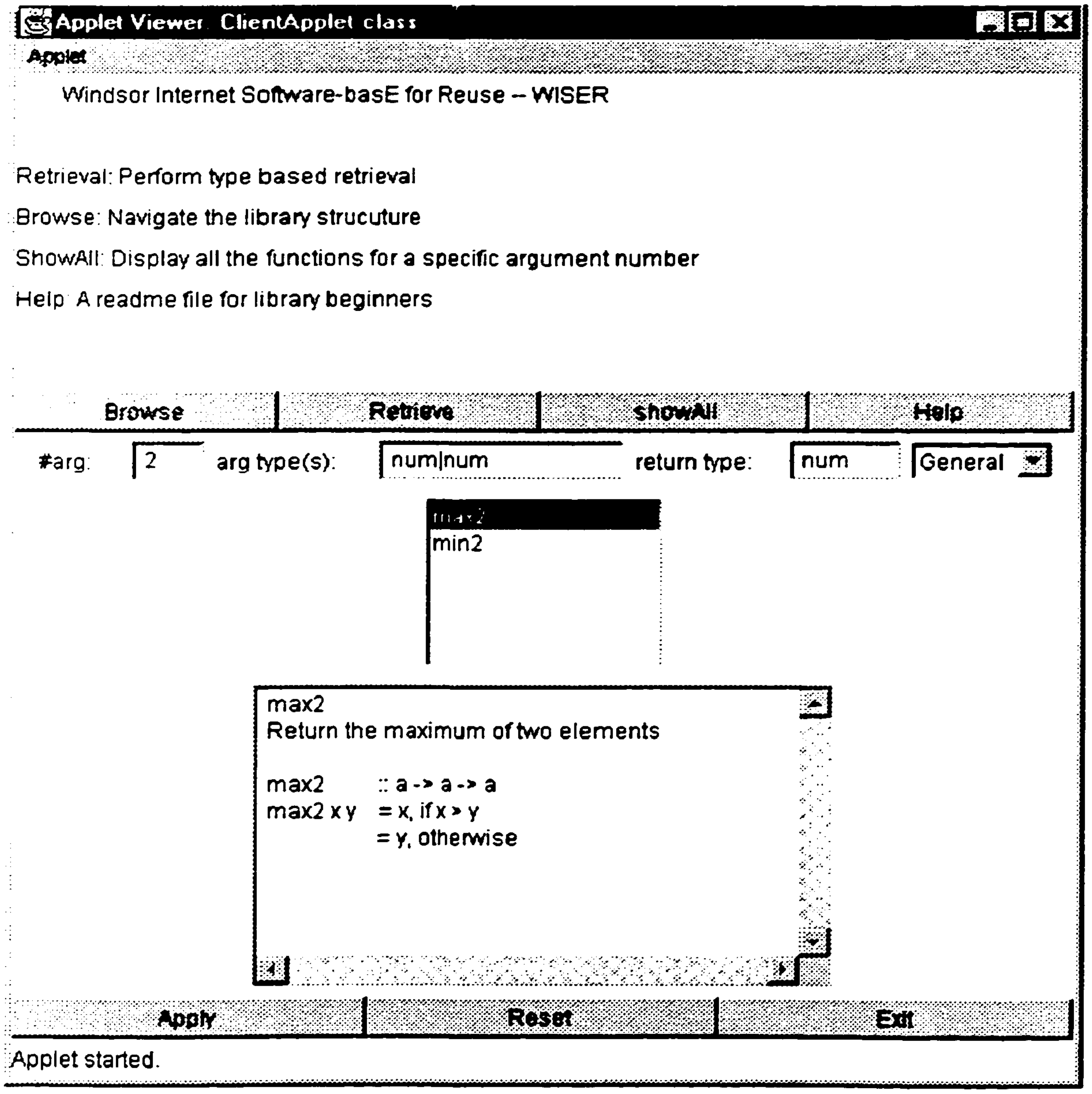
```
perm
Calculate the permutation

perm n r :: num -> num -> num
perm n r = fact n / fact (n - r)
```

Applet started.

If we want to find a function which returns the maximum of two numbers. We use ***num***

-> *num* -> *num* as search key. By performing exact match operation, we got a list of functions. By examining the source code for these functions, we find that none of them seems to match our requirement, as shown in Figure 38. If we perform general search,

Figure 39 General Match of *num -> num -> num*


Applet Viewer ClientApplet class

Applet

Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval

Browse: Navigate the library strucuture

ShowAll: Display all the functions for a specific argument number

Help: A readme file for library beginners

#arg: 2 arg type(s): num|num return type: num General

max2
min2

```
max2
Return the maximum of two elements

max2      :: a -> a -> a
max2 x y  = x, if x > y
          = y, otherwise
```

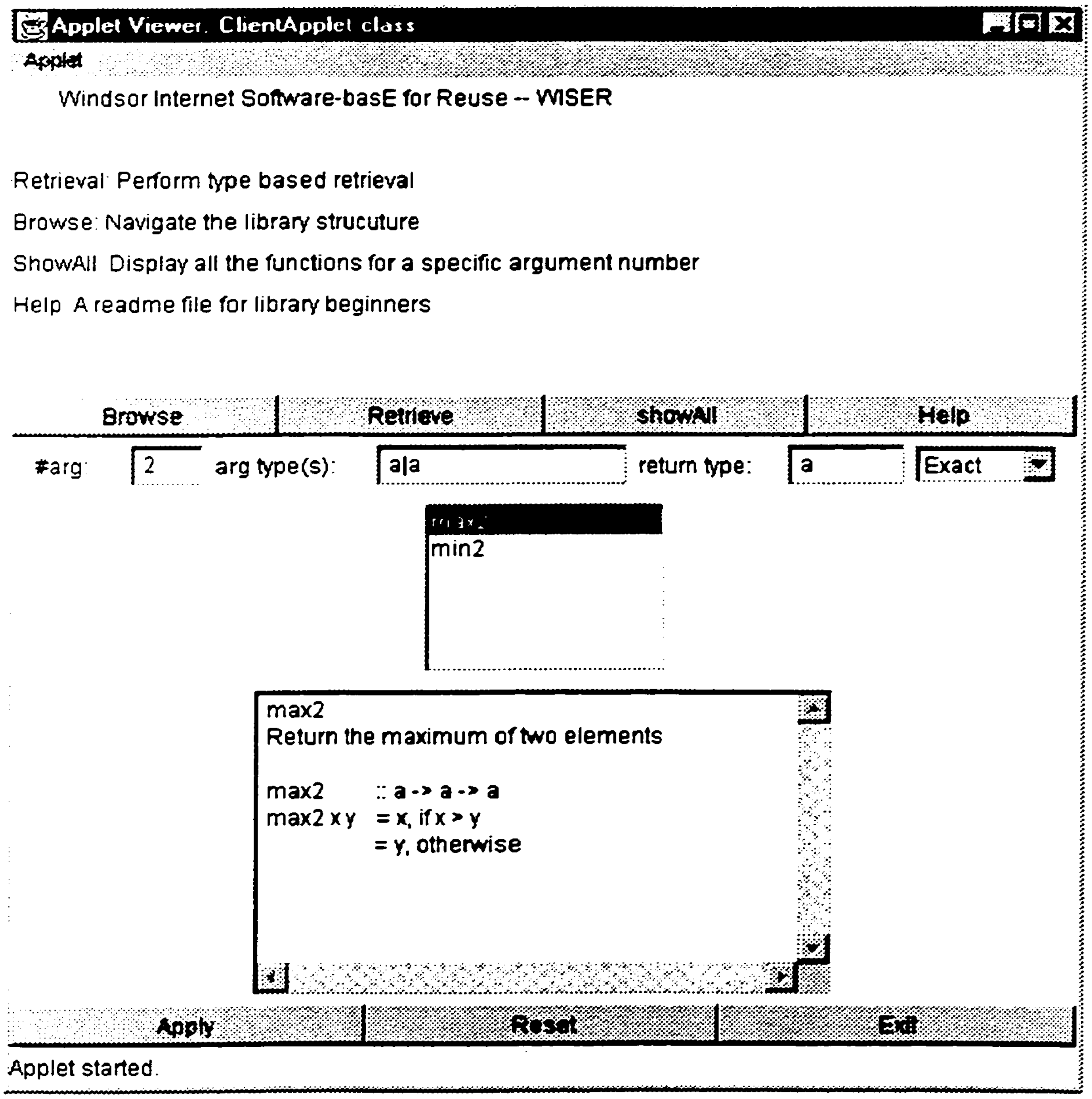
Applet started.

we find a function called "***max2***" which returns the maximum of two elements. If we replace the type ***a*** for "***max2***" with type ***num***, we can get our desired function very easily.

4.1.2.3 Specific Match

Suppose we want to retrieve a function which can be used to calculate the combinations of the given elements. We query **WISER** with ***a* -> *a* -> *a***, the exact matched functions "***max2***" and "***min2***" do not match our goal, as show in Figure 40.

Figure 40 Exact Match of *a* -> *a* -> *a*



Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval
Browse: Navigate the library strucuture
ShowAll: Display all the functions for a specific argument number
Help: A readme file for library beginners

#arg: 2 arg type(s): a|a return type: a Exact

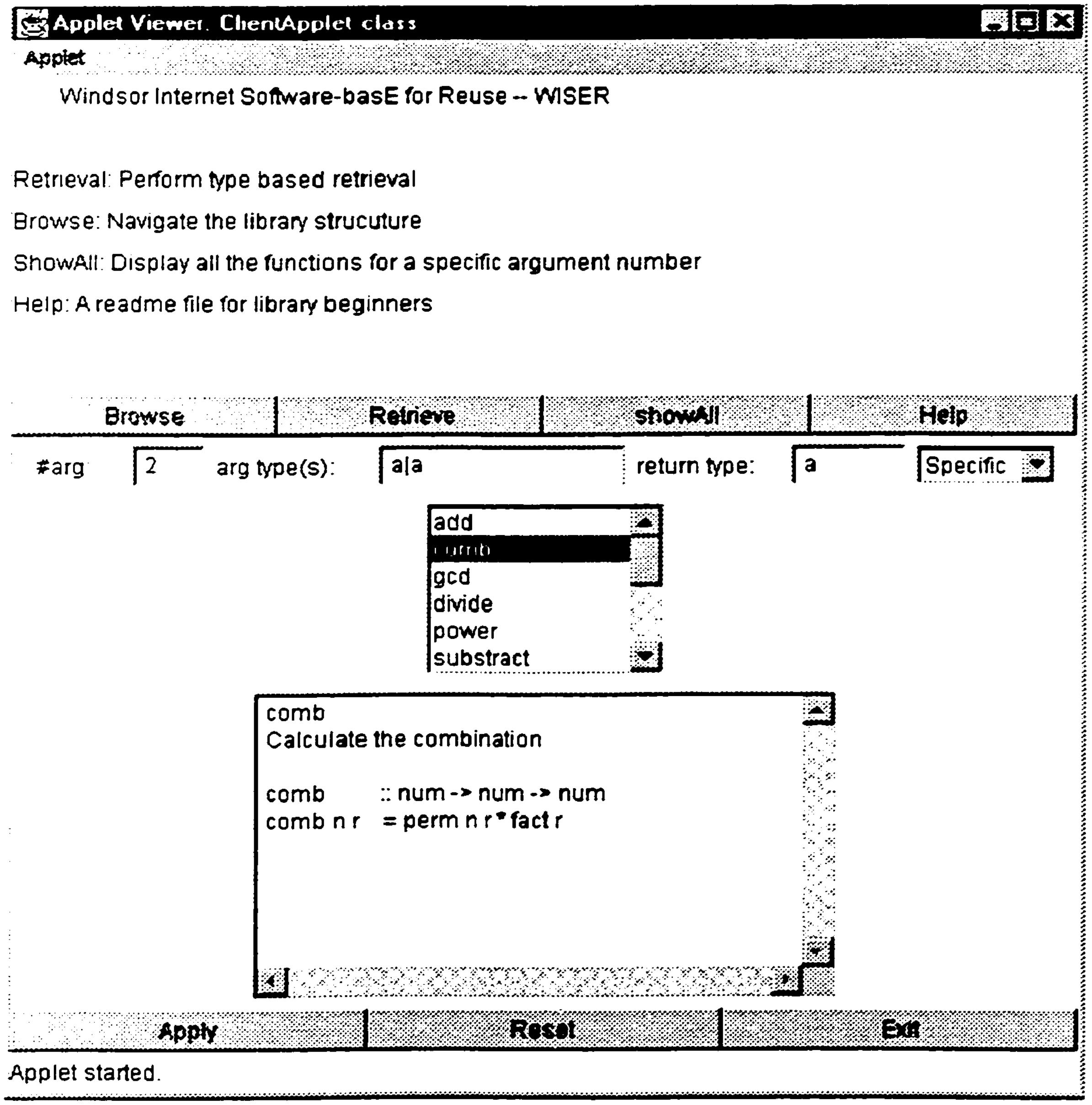
max2
min2

```
max2
Return the maximum of two elements

max2       :: a -> a -> a
max2 x y   = x, if x > y
           = y, otherwise
```

Applet started.

The specific matched function "***comb***" is similar as our goal, which performs combination over numbers, that is, the problem domain is smaller than our goal, as shown in Figure 41. By slightly modifying the source code (changing the type ***num*** to type ***a***), we can get our desired function.

Figure 41 Specific Match of *a* -> *a* -> *a*



Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval
Browse: Navigate the library strucuture
ShowAll: Display all the functions for a specific argument number
Help: A readme file for library beginners

#arg: 2 arg type(s): a|a return type: a Specific

add
comb
gcd
divide
power
substract

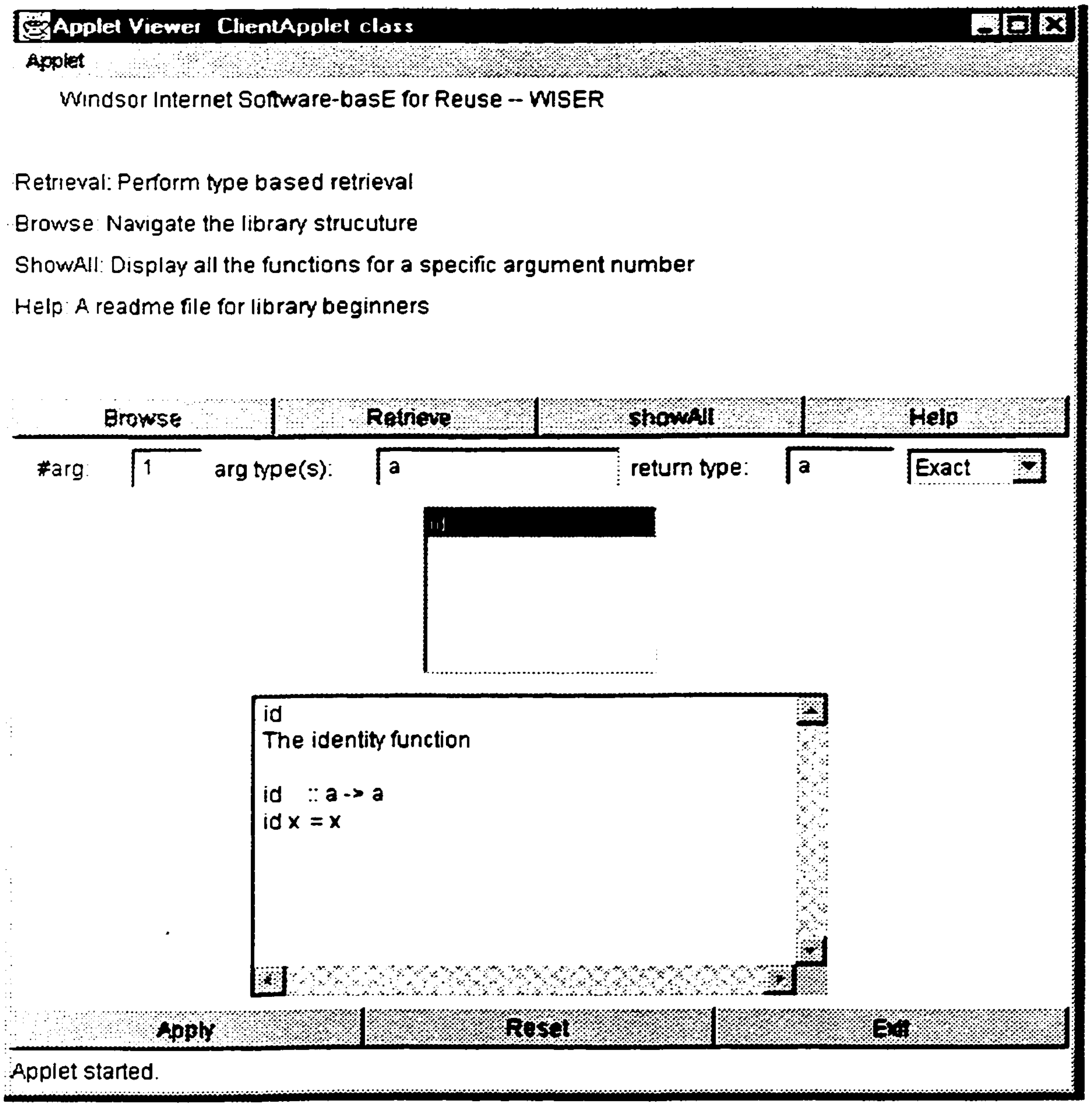
```
comb
Calculate the combination

comb        :: num -> num -> num
comb n r    = perm n r * fact r
```

Applet started.

4.1.2.4 Extra-Argument Match

Suppose we want to find a function which can create a constant valued function. We use *a* -> *a* to query **WISER** and perform exact match. The retrieved function "*id*" does not match our requirement. By performing extra-argument match, we got a function called "*const*", which is exactly what we want. Figure 42 and 43 illustrate this situation.

Figure 42 Exact Match of *a* -> *a*



Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval

Browse: Navigate the library strucuture

ShowAll: Display all the functions for a specific argument number

Help: A readme file for library beginners

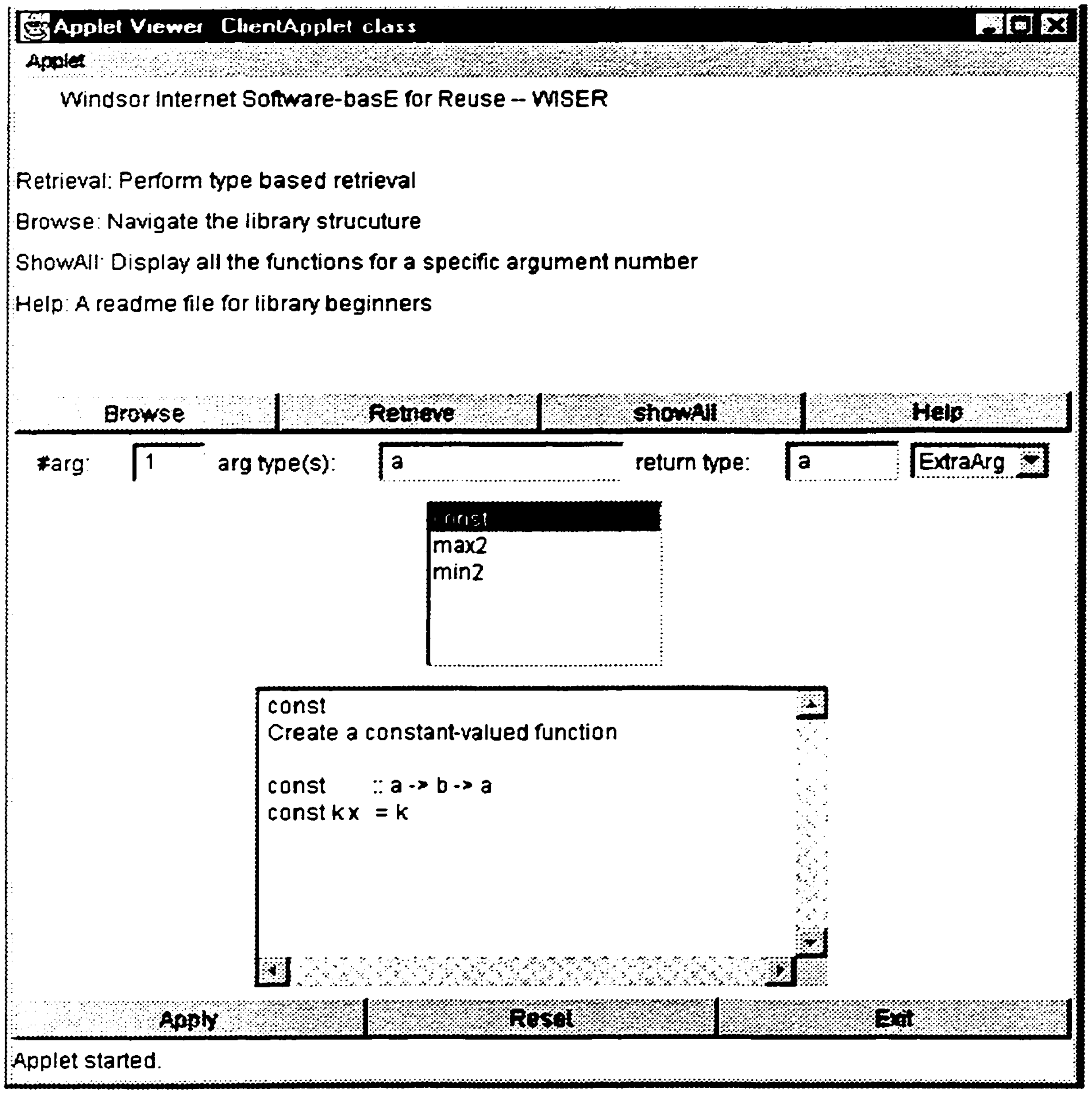
#arg: 1 arg type(s): a return type: a Exact

id

```
id
The identity function

id   :: a -> a
id x = x
```

Applet started.

Figure 43 Extra-argument Match of *a -> a*



Applet Viewer ClientApplet class

Applet

Windsor Internet Software-basE for Reuse -- WISER

Retrieval: Perform type based retrieval

Browse: Navigate the library strucuture

ShowAll: Display all the functions for a specific argument number

Help: A readme file for library beginners

#arg: 1 arg type(s): a return type: a ExtraArg

```
const
max2
min2
```

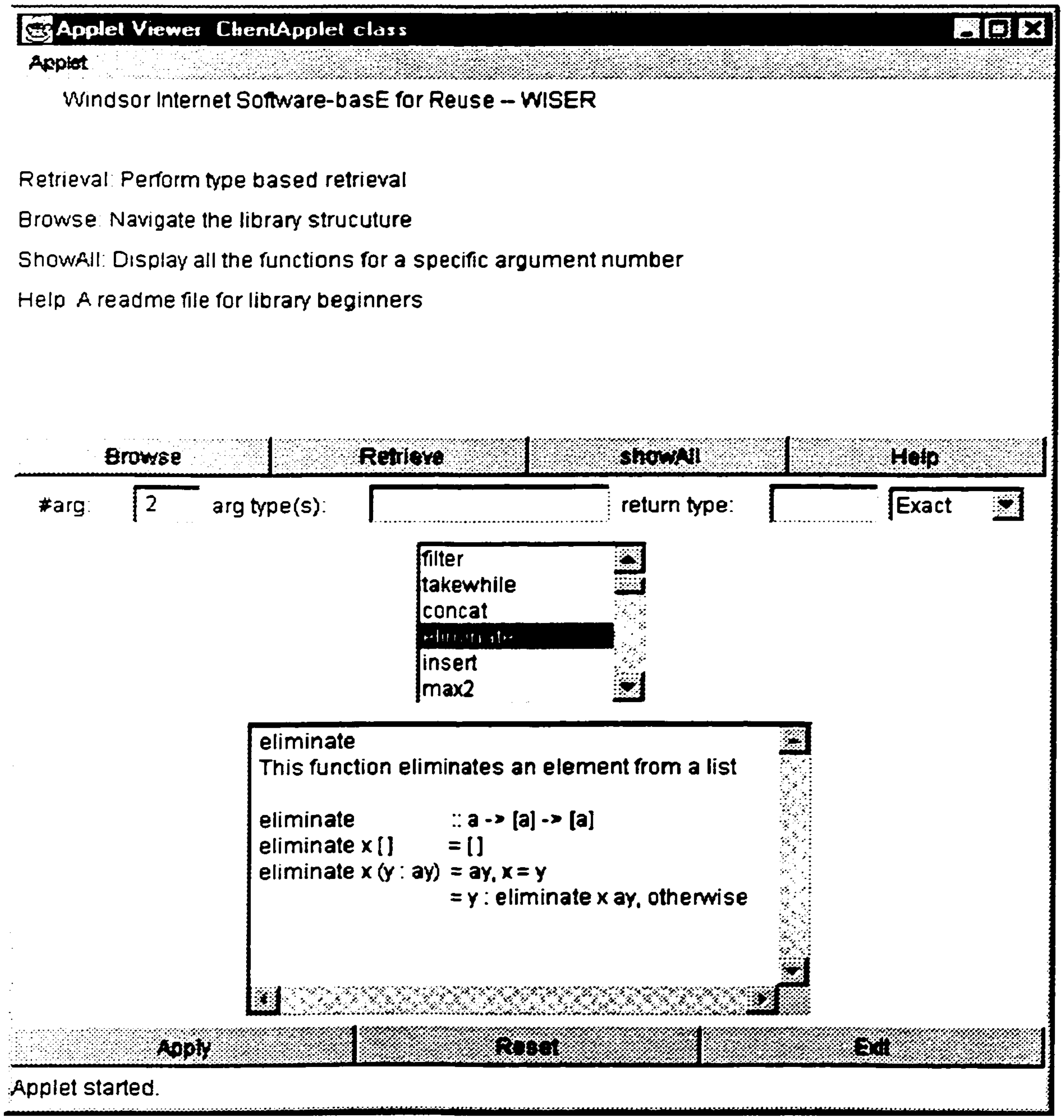
```
const
Create a constant-valued function

const      :: a -> b -> a
const k x  = k
```

Applet started.

4.1.3 Examples of showAll

"**showAll**" is used to display a list of function names in a particular sub-library. The user only needs to press "**showAll**" button and specifies the ***#arg***. Figure 44 is the ***showAll*** window ***for #arg = 2***. If the user selects a function from the list, say "***eliminate***", the source code of this function will be displayed in the *textArea*. The list holding the functions is scrollable.

Figure 44 ShowAll Window for #arg = 2



Windsor Internet Software-basE for Reuse – WISER

Retrieval: Perform type based retrieval

Browse: Navigate the library strucuture

ShowAll: Display all the functions for a specific argument number

Help: A readme file for library beginners

#arg: 2 arg type(s): return type: Exact

filter
takewhile
concat
eliminate
insert
max2

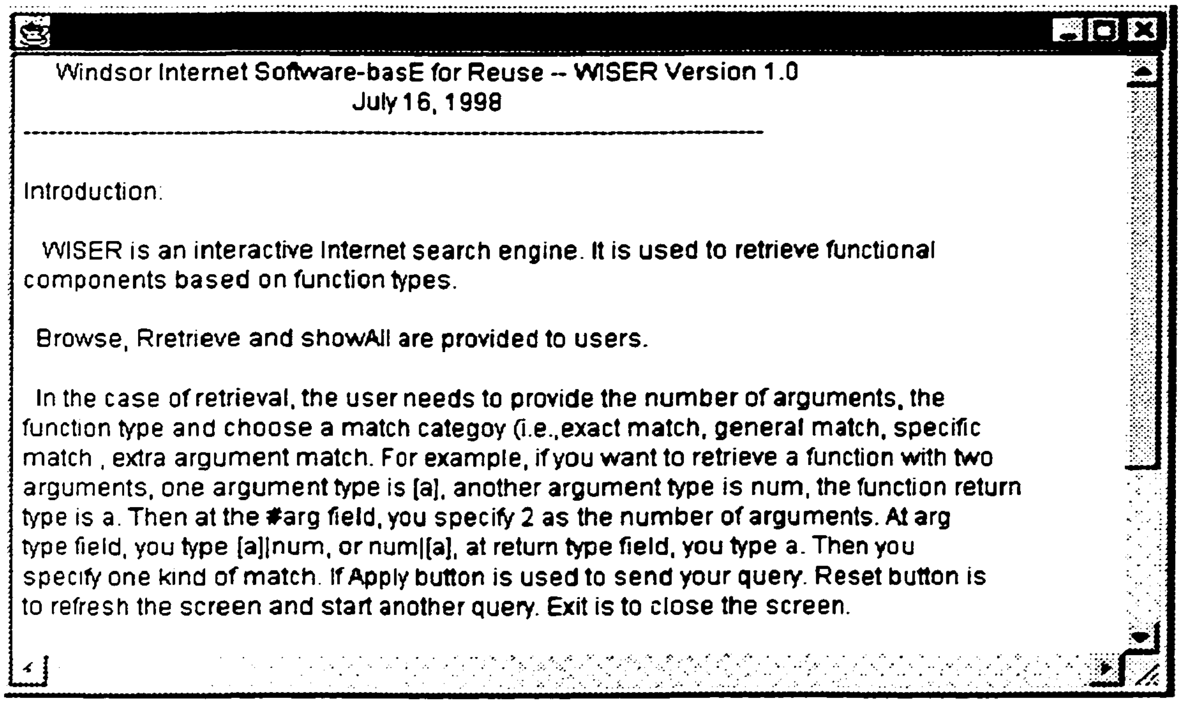
```
eliminate
This function eliminates an element from a list

eliminate              :: a -> [a] -> [a]
eliminate x []         = []
eliminate x (y : ay)   = ay, x = y
                       = y : eliminate x ay, otherwise
```

Applet started.

4.1.4 Help

A help window is designed to facility inexperienced users. It displays a README file, which describes detailed information about how to perform various operations over WISER. Figure 45 is the help window.

Figure 45 Help Window



Windsor Internet Software-basE for Reuse -- WISER Version 1.0
July 16, 1998

Introduction:

WISER is an interactive Internet search engine. It is used to retrieve functional components based on function types.

Browse, Rretrieve and showAll are provided to users.

In the case of retrieval, the user needs to provide the number of arguments, the function type and choose a match categoy (i.e.,exact match, general match, specific match , extra argument match. For example, if you want to retrieve a function with two arguments, one argument type is [a], another argument type is num, the function return type is a. Then at the #arg field, you specify 2 as the number of arguments. At arg type field, you type [a]|num, or num|[a], at return type field, you type a. Then you specify one kind of match. If Apply button is used to send your query. Reset button is to refresh the screen and start another query. Exit is to close the screen.

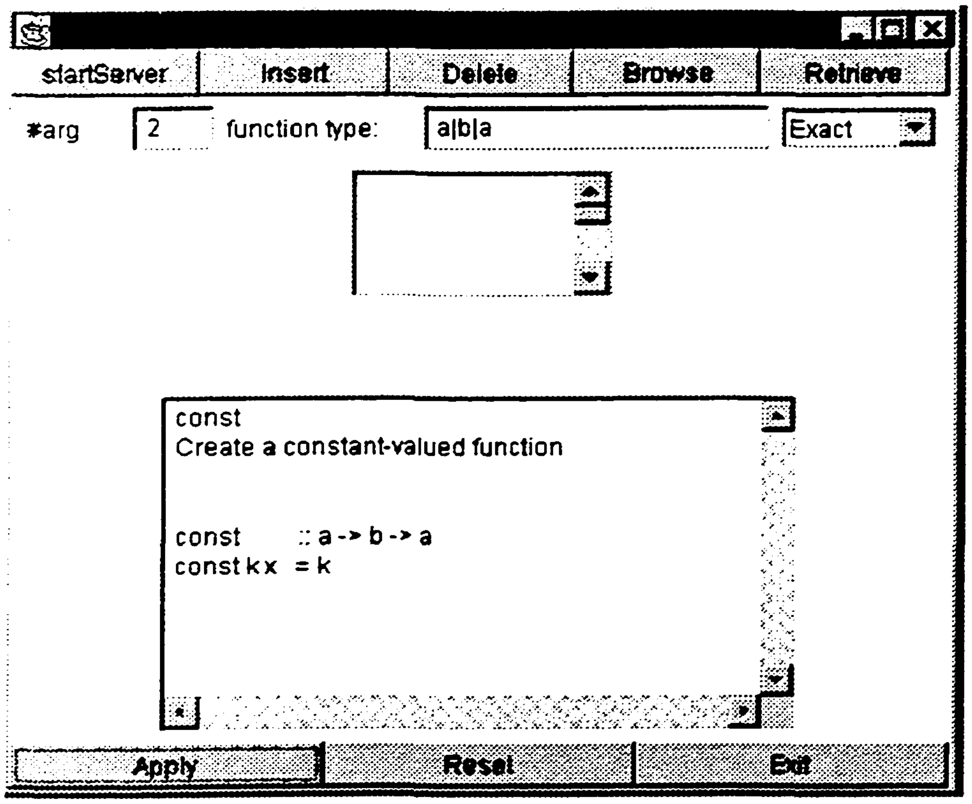
4.2 Server Side Interface

WISER is designed to allow only library developers to perform "**insert**" and "**delete**" operation. In order to investigate these two operations, let's take a look at the server side interface. The server side interface is similar as the lower portion of user interface except it contains different operations. "**StartServer**" is used to establish the socket connection between server side and user side. "**Insert**" is used to put function component into the library, "**Delete**" is used to take away existing function component from the library, "**Browse**" is used to list all the component names in a specified sub-library. "**Retrieve**" facilities the library developers to retrieve functions locally on the server side. These operations are based on the ***#arg*** and function type information. The function type represents all the argument types and return type delimited by "|" with the return type listed last (the argument types can be given in any order). A *ChoiceButton* is used to select exact match as well as flex matches for retrieval. Figure 46 is a sample server side interface. Let's use some examples to describe insertion and deletion in detail.

4.2.1 Examples of Insertion

Example1: The library now is empty, we first insert function "***const***" into the library. "***const***" is a two arguments function of type ***{a, b} -> a***. Figure 46 is the insertion window for "***const***". We specify ***#arg = 2***, **function type = *a|b|a***. The header part is the

Figure 46 Insertion of "const"



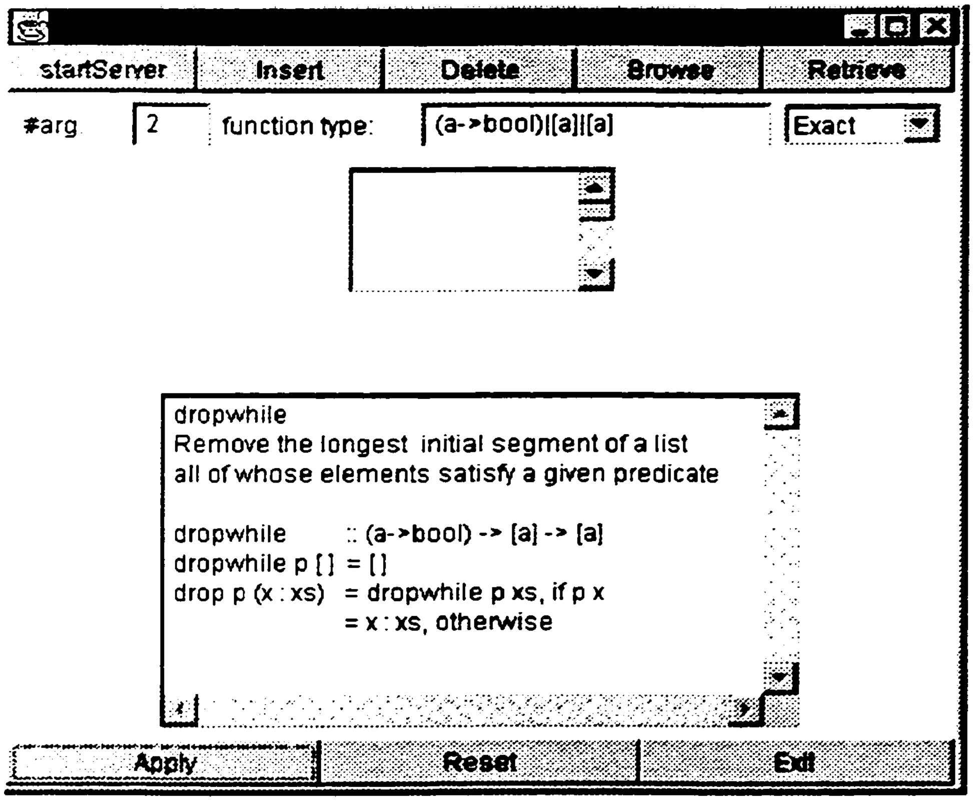
#arg 2 function type: a|b|a Exact

```
const
Create a constant-valued function


const      :: a -> b -> a
const k x  = k
```

function name (the first line in the *textArea*) and the explanation of the function. A blank line is used to separate source code part from the header part.

Example 2: We press "**Reset**" and continue to insert function "***dropwhile***" of type *{(a -> bool), [a]} -> [a]* into the library. We specify ***#arg* = 2**, *(a -> bool)\|[a]\|[a]* is the function type. Figure 47 is this insertion window.

Figure 47 Insertion of "*dropwhile*"



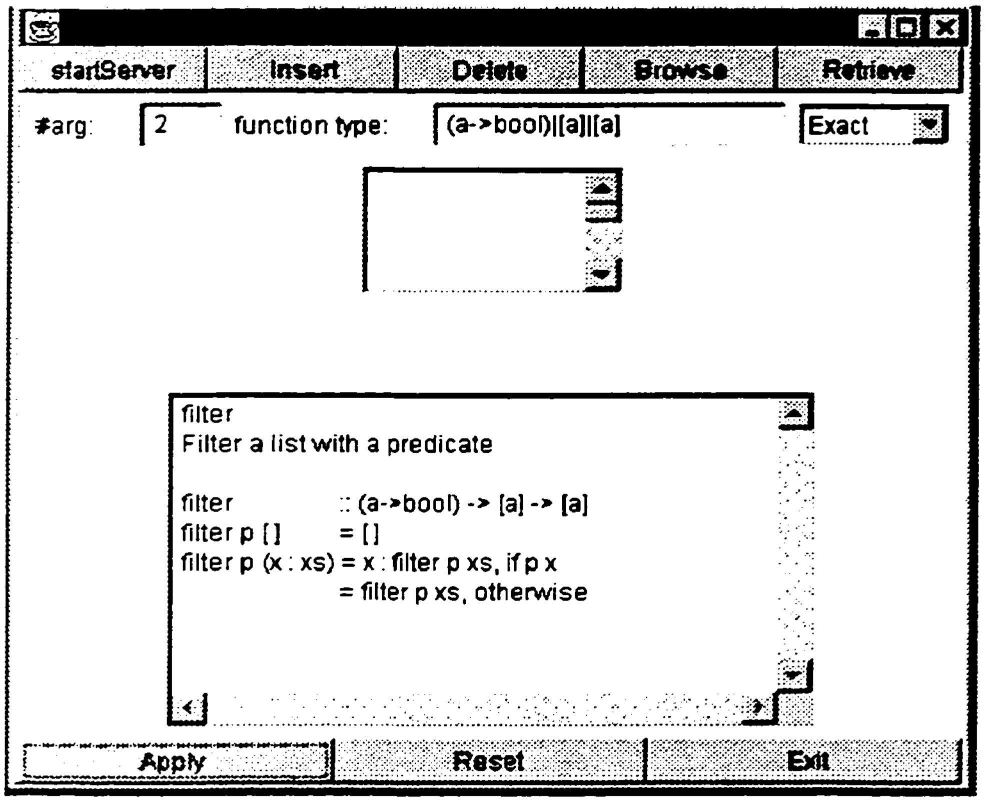
#arg 2 function type: (a->bool)|[a]|[a]

```
dropwhile
Remove the longest initial segment of a list
all of whose elements satisfy a given predicate

dropwhile         :: (a->bool) -> [a] -> [a]
dropwhile p [ ]  = [ ]
drop p (x : xs)  = dropwhile p xs, if p x
                 = x : xs, otherwise
```

Example 3: We continue to insert function "***filter***" of type ***{(a->bool), [a]}-> [a]*** into library. Figure 48 shows us the insertion window. We use "**Browse**" to confirm that function "***const***" is inserted into library. To do this, just click on "**Browse**" and specify

#arg* = 2**, and then press "**Apply**". Figure 49 shows us the window after we successfully insert function "const***", "***dropwhile***" and "***filter***".

Figure 48 Insertion of "*filter*"



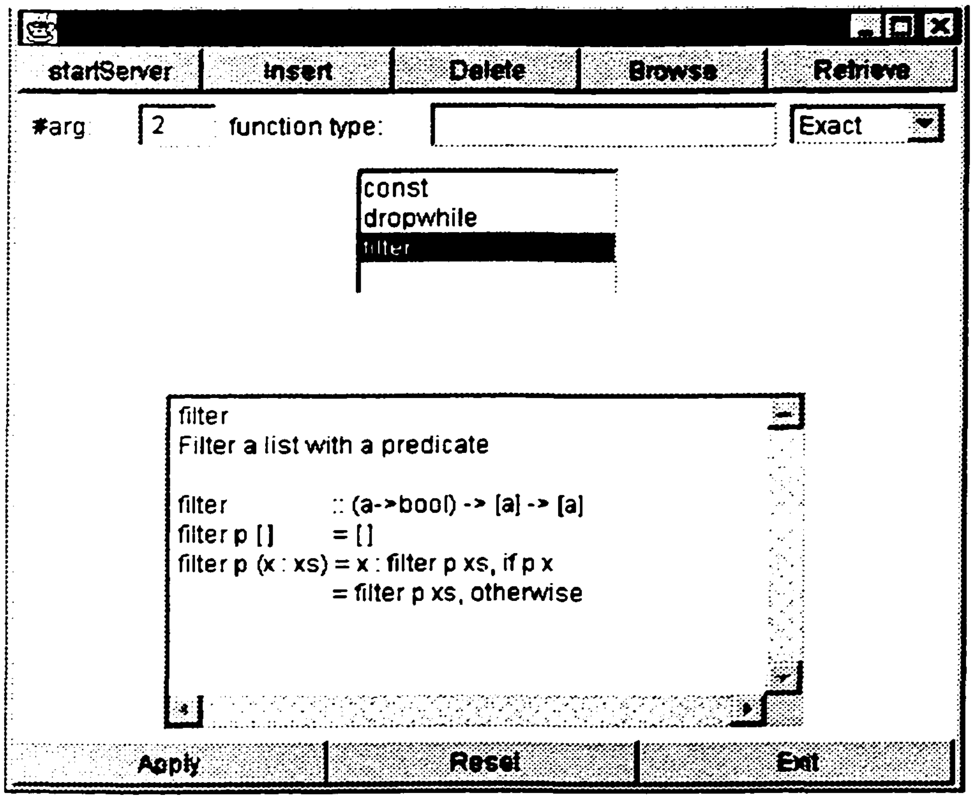
#arg: 2 function type: (a->bool)|[a]|[a] Exact

```
filter
Filter a list with a predicate

filter          :: (a->bool) -> [a] -> [a]
filter p []     = []
filter p (x : xs) = x : filter p xs, if p x
                = filter p xs, otherwise
```

4.2.2 Example of Deletion

Now we have three functions "***const***", "***dropwhile***" and "***filter***" in the library as shown in Figure 49. If we want to delete function "***dropwhile***" from the library, we first press the button "Delete", then specify ***#arg* = 2** and **function type** = ***(a -> bool)\[a]\[a]***, and then type the function name "***dropwhile***" in the first line of *textArea* and press return key. After "**Apply**" is pressed, function "***dropwhile***" will be deleted from the library. Figure 50 is the window for deletion of "***dropwhile***". Figure 51 is the confirmation window for the deletion of "***dropwhile***". Comparing Figure 49 and Figure 51, we can see that function "***dropwhile***" is no longer in the library any more.

Figure 49 Confirmation of Insertion of *"const"*, *"dropwhile"* and *"filter"*

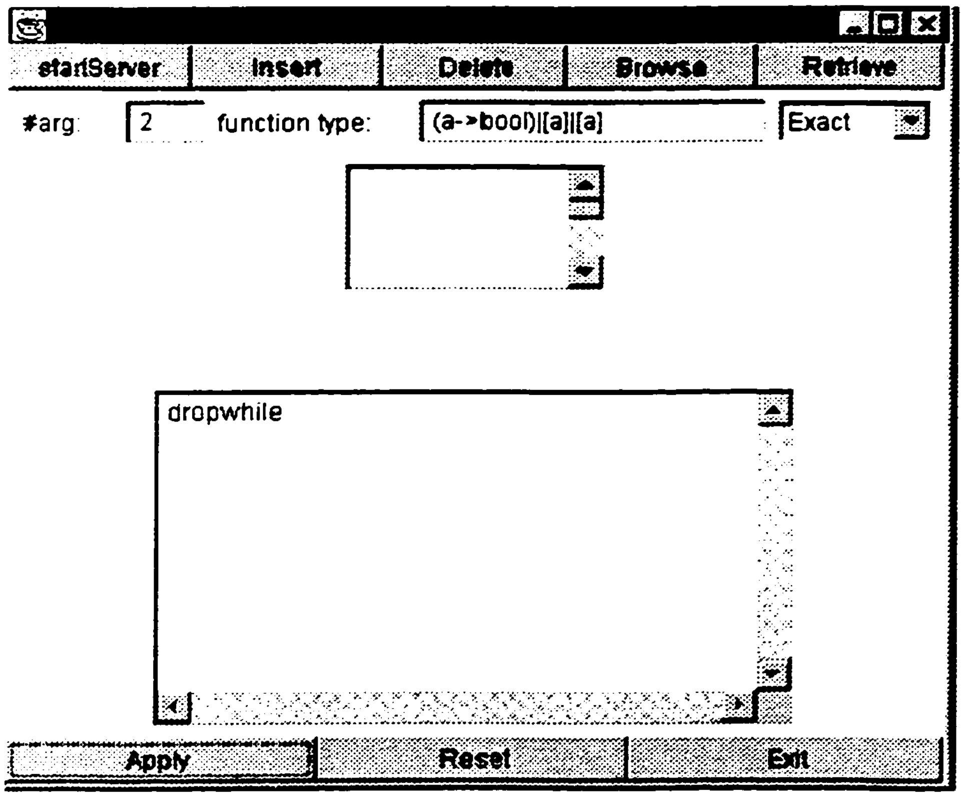


#arg 2 function type: Exact

const
dropwhile
filter

```
filter
Filter a list with a predicate

filter           :: (a->bool) -> [a] -> [a]
filter p []      = []
filter p (x : xs) = x : filter p xs, if p x
                 = filter p xs, otherwise
```

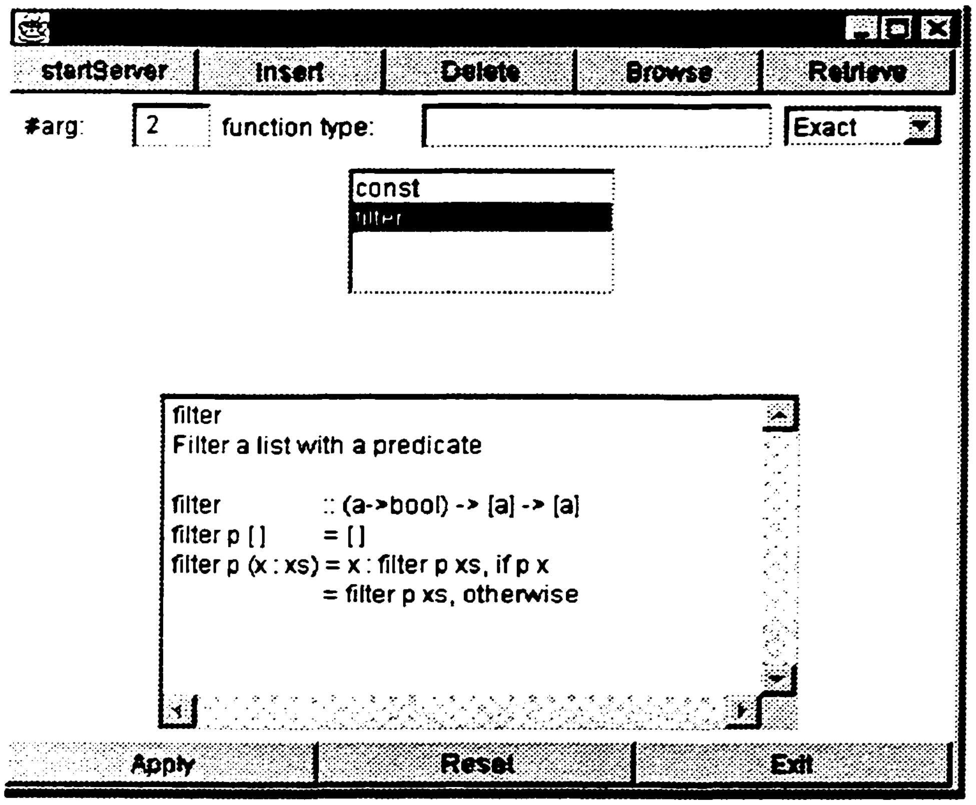
Figure 50 Deletion of *"dropwhile"*



#arg: 2 function type: (a->bool)|[a]|[a] Exact

dropwhile

Figure 51 Confirmation of Deletion of "*dropwhile*"



#arg: 2

```
filter
Filter a list with a predicate

filter          :: (a->bool) -> [a] -> [a]
filter p [ ]       = [ ]
filter p (x : xs) = x : filter p xs, if p x
                   = filter p xs, otherwise
```

Chapter 5 Conclusions and Future Work

This chapter summaries the conclusions of this thesis work and outlines the future work.

5.1 Conclusions

In this thesis we have investigated a structured software library to support reuse-oriented program development, and developed a prototype system called **WISER**.

The major purpose of this thesis is to maintain an evolving structured software base with various tools to allow for effective insertion, deletion, retrieval and browsing. This system also describes how the components in the library are grouped and linked by the number of arguments, the polymorphic and extra argument relations, thereby allow easy access to related functions. Because the users have no way to know the order of arguments associated with functions in the library, **WISER** is designed to ignore the difference of the argument order by using a new module based on **set_type** concept. A breadth-first like algorithm is used to search functions. All the user's interactions with **WISER** are Window-based so as to allow easy input/output. The complexity of the retrieval system proves to be linear in terms of the number of types for the worst case. In average it is much better.

Based on the work carried out in this thesis we have the following conclusions:

1. Using **WISER** with its structured software library based on function types disregarding the order of arguments provides a practical approach for reuse-oriented program developments, especially for reuse in the large.
2. Type-based retrieval method is not precise but can give a big cut over the candidate components. This method can be integrated with other method to achieve more precise result.

3. **WISER** supports automatic retrieval.
4. **WISER** supports both exact match and relaxed matches.
5. **WISER** can be posted to WWW. All the users can access it from remote area by using the browsers like Netscape.
6. The average performance is better than sequential search.
7. A user-friendly, easy-to-use interface is provided.
8. **WISER** is an open system and can grow to be very large.
9. **WISER** can be used as an aid in functional programming coding phase.

5.2 Future Work

Retrieval from a reuse library based on type information using our approach can get suitable candidate functions. However, the number of components retrieved is unpredictable. So the accuracy of this method is low. Therefore this approach needs to be complemented with other techniques so as to narrow down the candidate functions to the most appropriate one(s).

This structured approach with the argument order disregarded can be extended to other reuse area, for example retrieval based on functional composition, retrieval on other programming components, such as classes in Object-Oriented programming environment.

APPENDIX

One argument random access file (arg1.dat)

```
{a|b}{ }{ }{ }{0}

{[a]|num}{lenlist}{0}{320}{ }

{[num]|num}{avgr|product|sum}{160|480}{ }{ }

{[a]|a}{hd|last|max|middle|min}{0}{320|640|1760}{1440|2560}

{[[a]]|[a]}{listConcat}{480}{ }{ }

{a|a}{id}{0}{960|1120|2400}{160|640}

{num|num}{fact|fib|lstdiv}{800}{ }{800}

{[a]|[a]}{boolSum|elizero|init|sort|reverse|tl}{800}{1440}{320|480|1120|1280}

{num|[num]}{divisors|pridiv|upto}{0}{ }{2400}

{[num]|[num]}{sieve}{1120}{ }{ }

{(a,b)|a}{fst}{0}{ }{ }

{[bool]|bool}{and|or}{480}{ }{ }

{num|bool}{even|odd|prime}{0}{ }{ }

{num|[(num, num, num)]}{triads}{0}{ }{ }

{([a],[b])|[(a,b)]}{zip}{0}{ }{ }

{bool|bool}{negation}{800}{ }{960}
```

Two arguments random access file (arg2.dat)

```
{a|b|c}{ }{ }{ }{0}

{a|b|a}{const}{0}{320|480|640|1280}{ }
```

```
{(a->bool)|[a]|[a]}{dropwhile|filter|takewhile}{160}{ }{ }

{a|[a]|[a]}{concat|eliminate|insert}{160}{ }{ }

{a|a|a}{max2|min2}{160}{800|960|1120}{ }

{num|num|num}{add|comb|gcd|divide|power|substract|times|perm}{640}{ }{ }

{bool|bool|bool}{disjun|conjun}{640}{ }{ }

{[a]|[a]|[a]}{concatlist|merge|difflist}{640}{ }{ }

{num|[a]|[a]}{drop|take}{160}{ }{ }

{(a->a->a)|[a]|a}{foldr1|foldl1}{0}{ }{ }

{(a->b)|[a]|[b]}{map}{0}{ }{ }

{(a->a)|a|[a]}{iterate}{0}{ }{ }

{a|a|bool}{less|lessequ|greatequ|greater|notequ}{0}{ }{ }

{[(a,[char])]|a|[char]}{assoc}{0}{ }{ }

{[a]|a|bool}{member}{0}{ }{

{num|num|[num]}{comdiv}{0}{ }{ }

{[a]|num|a}{indexlist}{0}{ }{ }
```

Three arguments random access file (arg3.dat)

```
{a|b|c|d}{ }{ }{ }{ }

{(a->b->a)|a|[b]|a}{foldl}{0}{ }{ }

{(a->b->b)|b|[a]|b}{foldr}{0}{ }{ }

{(a->b->a)|a|[b]|[a]}{scan}{0}{ }{ }

{(a->bool)|(a->a)|a|a}{until}{0}{ }{ }
```

BIBLOGRAPHY

[1] Krueger, W., (1992) "Software reuse", *ACM Computing Surveys*, 24(2):131-183

[2] Cheng, J., (1994) "A reusablility-based software development environment." *ACM SIGSOFT, Software Engineering Notes*, 19(2): 57-62.

[3] Cheng, B. and Jeng, J. (1992) "Formal methods applied to reuse", *Proceedings of the Annual Workshop on Software Reuse.*

[4] Podgurski, A and Pierce, L. (1993) "Retriving reusable software by sampling behavior", *ACM Transaction on Software Engineering and Methodology*, 2(3): 286-303.

[5] Zaramski, A. and Wing, J. (1993) "Signature Matching: A Key to Reuse." *Proceedings of the ACM SIGSOFT Symposium on Foundation of Software Engineering*, pp. 182 – 190.

[6] Zaramski, A. and Wing, J. (1995) "Specification Matching of Software Components", *Proceeding of the ACM SIGSOFT Symposium on Foundation of Software Engineering.*

[7] Rollins, E. and Wing, J. (1991) "Specifications as Search Keys for Software Libraries", *Proceeding of the International Conference on Logic Programming.*

[8] Bird, R. and Walder, P. (1988) "Introduction to Functional Programming." *Prentice Hall.*

[9] Park, Y. and Ramjisingh, D. (1995) "Software Component Base for Reuse in Functional Program Development." *Proceedings of the International Conference on Computing and Information*, pp. 1022 – 1039

[10] Rittri, M. (1991) "Using Types as Search Keys in Function Libraries." *Journal of Functional Programming*, 1(1): 71 – 89

[11] Rittri. M. (1990) "Retrieving Library Identifiers via Equational Matching of Types." In M. E. Stickel. editor, *Int. Conf. On Automated Deduction, vol.449 of Lecture Notes in Artificial Intelligence*, pp. 603-617, Springer-Verlag.

[12] Rittri. M. (1993)"Retrieving Library Functions by Unifying Types Modulo Linear Isomorphism." *Chalmers University of Technology and University of Goteborg.*

[13] Runciman. C. and Toyn, I. (1991) "Retrieving Reusable Software Components by Polymorphic Type." *Journal of Functional Programming* 1 (2): 191 – 211.

[14] Zand. M. H.and Samadzadeh. M. K., (1994) "Software Reuse: Issues and Perspectives", *IEEE Potentials for Engineers*, August/September 1994, pp. 15 – 19.

[15] Ramjisihgh D., (1994) "Software Base for Reuse-oriented Program Development", *Master thesis*, University of Windsor.

[16] Bai. P., (1995) "Execution-based Retrieval of Reusable Software Components", *Master thesis*, University of Windsor.

[17] An. Q., (1997) "Software Reuse in Object-oriented Programming", *Survey of Background Reading*, University of Windsor.

[18] Mili. A., Mili, R. and Mittermeir, R. (1994) "Storing and Retrieving Software Components: A Refinement-based approach", *Proceeding of the International Conference on Software Engineering.*

[19] Marrek. Y., Berry, D. and Kaiser, G. (1991) "An Information Retrieval Approach for Automatically Constructing Software Libraries", *IEEE Transactions on Software Engineering*, 8(17), 800 – 813.

[20] Prieto-Diaz. R. and Freeman. P. (1987) "Classifying Software for reusability", *Software*, pp6-16.

VITA AUCTORIS

Qiuyan An (Nancy) was born in 1965 in **Beijing, China**. She graduated from **Dayu High School** in 1984. From there she went on to **Tsinghua University** where she obtained a Bachelor degree in Electrical Engineering in 1989. She is currently a candidate for the Master's degree in Computer Science at the **University of Windsor** and hopes to graduate in the Fall of 1998.