Signature and Specification Matching

Amy Moormann Zaremski

January 1996

CS-CMU-96-103

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted to Carnegie Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

Thesis Committee:
Jeannette M. Wing, Chair
David Garlan
Peter Lee
Steven J. Garland, Massachusetts Institute of Technology

This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the U. S. Government.

Keywords: software reuse, software libraries, component retrieval, library indexing, subtyping, signature matching, specification matching

Abstract

Large libraries of software components hold great potential as a resource for software engineers, but to utilize them fully, we need to be able: (1) to locate components in the library; (2) to organize the library in a way that facilitates browsing and improves efficiency of retrieval; and (3) to compare the description of a library component to the description of what we want.

A key requirement in all of these problems is to be able to compare two software components to see whether they *match*. In this dissertation, we consider two different kinds of *semantic* descriptions of components to determine whether components match: *signatures* (type information) and *specifications* (behavioral information). Semantic descriptions offer advantages over either textual descriptions, such as variable names, or structural descriptions, such as control flow graphs. Using semantic information focuses on *what* the components do rather than how they do it. Signatures and specifications are natural ways of describing software components and have well-understood properties, such as type equivalence and logical relations between formal specifications, that enable us both to define matches precisely and to automate the match.

This dissertation makes the following contributions:

- *Foundational.* Within a general, highly modular, and extensible framework, we define matching for two kinds of semantic information (signatures and specifications) and two granularities of components (functions and modules). Each kind of matching has a generic form, within which all of the matches are related and may, in some cases, be composed. The orthogonality of the matches allows us to define match on modules independently of the particular match used on functions in the modules.

- *Applications.* We show how the definitions of matching can be applied to the problems of retrieval from libraries, indexing libraries, and reuse of components. We demonstrate the various signature and specification matches with examples of typical uses in each application.

- *Engineering.* We describe our implementations of function and module signature match, function specification match, function signature-based indexing, and function signature-based retrieval. These implementations demonstrate the feasibility of our approach and allow us to illustrate the applications with results from a moderately-sized component library.

Acknowledgements

I owe a deep debt of gratitude to my advisor, Jeannette Wing, for both her technical and moral support. Jeannette is an endless source of information and insights on technical issues and has a way of asking the questions that open up new angles on a topic. She is also able to sense somehow whether I need support and encouragement or a stern "nudge" in the right direction. I would never have made it without her.

I would like to thank the other members of my thesis committee, David Garlan, Peter Lee, and Steve Garland, all of whom have given me invaluable assistance, both with insights on the "big picture," with technical details, and with excellent comments on an earlier draft of the thesis.

A number of people at Carnegie Mellon helped me over the years. I am thankful to all the members of the Miró, Venari, and Coda projects, as well as members of the Composable Systems group. I particularly want to thank Gene Rollins, for his help in modifying the SML compiler to provide the necessary hooks for the signature matcher, and Chris Okasaki, for his helpful suggestions both on an early draft of the thesis and in a number of discussions, particularly about the implementation to build indexed libraries. Both Gene and Chris also tested Beagle and suggested improvements. I also want to thank Maria Ebling, not only for helping me keep Coda running on my machines so that I could write at home, but for being such a good friend.

I also owe thanks to all the great people at Carnegie Mellon who keep things running smoothly and who cheerfully make all the administrative hassles go away, especially Sharon Burks, Catherine Copetas, Cary Lund, and everyone on the facilities staff.

I would also like to thank all those who have helped keep me sane throughout this process, in particular Maria, Chris, Francesmary Modugno, Mark Maimone, the members of CTQC, and all my other friends. A special "woof" of thanks to my basset hounds, Raleigh[1] and Augie[2], who have been quite happy to assist me in the quest for acronym appendages to names.

I extend my heartfelt thanks to my husband, Mark Zaremski, for his love, support, patience, and faith in me. I especially thank him for understanding and tolerating the irrationality of "thesis mode". He and I are two halves of a whole – without him I would be far less than I am.

Finally, I would like to thank my parents for the uncountable ways they have helped me, and in particular for always believing that I could do anything. My father, Ralph Moormann, taught me to question and to be curious, and my mother, Frances Moormann, showed me the meaning of determination. Those two elements have combined to give me the interest and capability to do research in science.

[1] Blu-line Raleigh, CGC, TDI

[2] Salvador D'Augie, CGC, ILP pending

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Demand for software continues to increase, and software systems continue to grow in size and complexity. The challenge for software engineering is to meet these demands as cheaply and as quickly as possible. Software libraries hold great potential as a resource for the software engineer, both to enable him or her to reuse existing software components to build larger systems and as a source of examples to become more familiar with a language or with a style of programming usage. There is a growing collection of software libraries, especially on the World Wide Web.

Reusing existing components can decrease the time spent building a large system (and thus decrease costs), since it can significantly reduce the amount of new code that must be written. Furthermore, reusing components from a well-tested library can reduce time spent debugging and improve reliability. Components in software libraries are also more likely to have been verified formally. Time and expense spent on verification is more easily justified for library components than for code used only once, since such costs can be amortized over multiple uses.

The first challenge in reuse is to be able to locate a component in the library. It should be faster and easier to find a component than to write it from scratch. We can find a component either by describing it with a query, and *retrieving* components that match the query, or by *browsing* through the library (preferably indexed somehow) until we find a component we want. Once we have found a component, we must be able to *compare* it to the task at hand. We may be able to use the component directly, or may have to modify it slightly.

The activities of retrieving, browsing, and comparing have other uses as well. For example, software libraries are a source of examples of the use of a programming language. A software engineer can learn how to use particular language constructs and learn about the style of programming for the language, either by browsing the library, or retrieving particular components. Retrieving components from a library can also provide statistics about the contents of the library, such as what percentage of the functions in a library have more than one input. The same index structure used to enable browsing on the library can be used to improve efficiency of retrieval. And comparing two components answers the general question of how they are related, even outside the context of reuse. For example, we can determine whether one component is a subtype of another.

In this thesis, we present a way to retrieve components from a library, index a library, and compare two components, to help realize more of the potential of software libraries. We use semantic information about software components to do this. In particular, we assume that each component in a library has associated with it a *signature* (type information) and possibly a *specification* (behavioral information). In the remainder of this chapter, we describe in detail the problems we want to handle (Section 1.1), the approach we use in the thesis to solve the problems (Section 1.2), the main contributions of the thesis (Section 1.3), and a roadmap to the rest of the thesis (Section 1.4).

The general approach of using semantic descriptions of components applies to many other domains in addition to software. For example, consider the information available through the World Wide Web. Having an effective way to describe the kinds of information we are interested in could simplify the increasingly daunting task of locating the information we want. Other examples of domains where semantic information could aid in retrieval include the nationwide Library of Congress, law briefs, police records, and geological maps.

1.1 Problem Description

Consider the following list of seemingly diverse questions:

1. *Retrieval.* How can I retrieve a component from a software library based on its semantics, rather than its syntactic structure?

2. *Indexing.* How can I index the components in a software library?

3. *Navigation.* Given a hierarchical index of a library, let me see all the nodes one level up from the current component.

4. *Substitution.* When can I replace one software component with another without affecting the observable behavior of the entire system?

5. *Subtyping.* When is one type a subtype of another?

6. *Modification.* How might I adapt a component from a software library to fit the needs of a given subsystem?

Each of these questions is interesting on its own. Most work on software reuse cites the crucial problem of retrieving a component from a library (question 1) [AM87, BP89, Kru92, MMM95, IEE84]. We cannot reuse a component if we cannot find it. Adding an index to a library (question 2) enables us to navigate through the library (question 3) and increases the efficiency with which we can store and retrieve components. The advantages of hierarchical indexes are well-understood in the object-oriented domain, where users can navigate the class inheritance structure with a browser (e.g., Smalltalk [Tes81] and C++ [Bis92]). However, there has been no previous work to extend the use of indexes beyond object-oriented languages.

Currently many libraries use the file system for their only organization (directories and files) and file system and editor commands for navigation and retrieval. For example, the local ML library is organized with categories of components as directories (e.g., `local/lib/Container/`, `local/lib/Threads/`); users locate desired components with Unix tools such as `ls` and `grep`. Aside from some of the information that could be gleaned from how the library is organized, the task of finding something in these libraries relies on the names of components. Sharing components with others requires mutual agreement on a naming scheme and a directory structure.

Once we have retrieved a component from a library, there is the additional issue of how to use it. Can we substitute it directly where we need a component, or do we have to modify it in some way before we can use it? If we substitute directly, we would like to know that the component behaves in a way we expect (question 4). That is, if we have specified the behavior we expect, we would like to know that the component behaves in a way consistent with that specification. A special kind of substitution is the notion of subtyping in object-oriented languages (question 5). Defining when one component is a subtype of another, particularly behaviorally, is a current research topic of interest [Ame91, Car89, DL92, Lea89, LW90, LW94, Mey88]. If we cannot substitute a component directly, we need to know how to adapt it for reuse in the current context (question 6). Knowing how a retrieved component differs from what we want (i.e., identifying a mismatch) can often help determine how to modify the component.

The questions listed above also share some commonalities. In retrieval, we search for all library components that *satisfy* a given query. In building a hierarchical index on a library, we *relate* each pair of components. In navigation, we go from one component to another that is *higher* (or lower) in the hierarchy. In substitution, we expect the behavior of one component to be observably *equivalent* to the other's; a special case is substituting a subtype object for a supertype object. In modification, we adapt a component to fit its environmental constraints, based on how well the component *meets* our requirements. Common to answering these questions is deciding when one component *matches* another, where "matches" generically stands for "satisfies," "relates," "is higher," "is equivalent to," or "meets."

1.2 A Solution (Thesis Scope)

In this thesis, we define various kinds of *component matching*. Most generally, a component match function, M, takes two components (or abstract descriptions of the components) and returns a boolean indicating whether a particular relation holds between the two components.

Declaration 1.2.1[1] (Component Match)

M : Component $*$ Component $\rightarrow$ Bool

We vary three parameters in our concept of match: the kind of information used to describe the components, the granularity of the components, and the degree of relaxation of the match. Figure 1.1 illustrates the design space created by these three parameters.

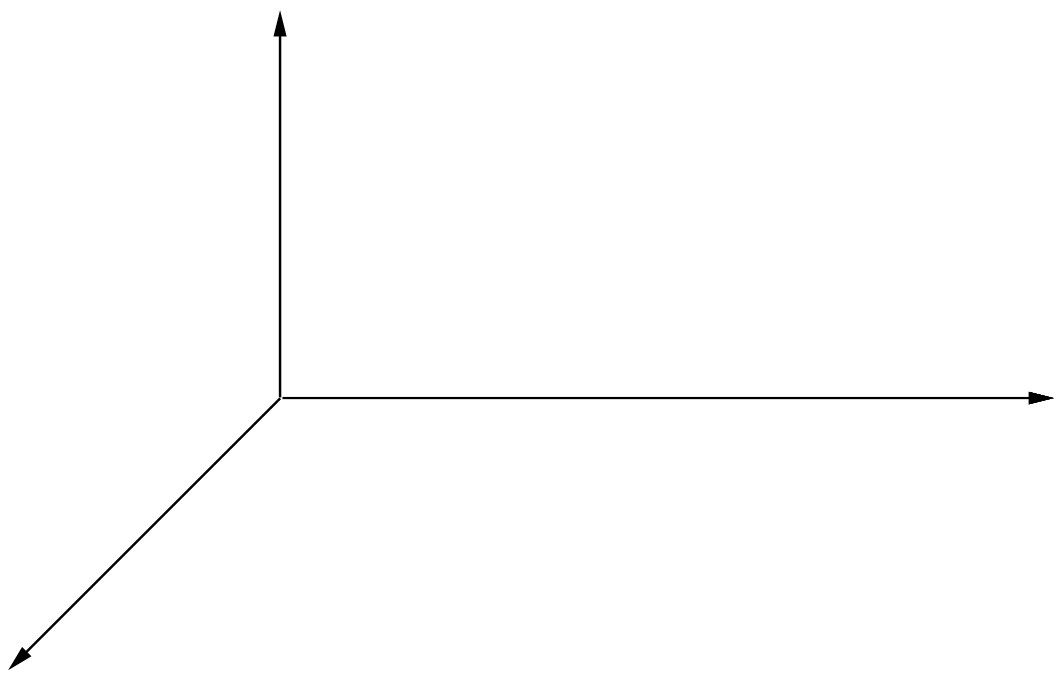


Figure 1.1: Design space of component match

The x-axis indicates the kind of abstraction used by the match. Components themselves may be either textual pieces of code or executable binaries. In either case, it is unlikely that components themselves will be the same, or even similar enough to relate. Therefore, we must compare *abstracts* of components. An abstract of a component is a description of the component that eliminates some of the details or that characterizes the component at a more abstract level. Examples of component abstracts include textual descriptions, structural information, signatures, and specifications. The semantic richness of abstracts increases as we move to the right on the x-axis.

With textual descriptions, matching is based on text strings, and perhaps some knowledge about synonyms. There is usually at least some textual information available about any com-

[1] *Declarations* declare a function and its type; *Definitions* include a definition of the function as well.

ponent (e.g., variable names, documentation). A drawback of textual descriptions, however, is the lack of precision. For example, what does "delete" mean? What format does the returned value have? And how can we define matching when people use very different words to describe similar concepts, or perhaps even a different language?

Examples of structural abstracts include control flow or data flow graphs. Although such graphs are precise, the abstracts they provide focus on *how* the component works, rather than *what* the component does. The questions we want to answer are more interested in the what, not the how.

Semantic abstracts enable us both to describe the behavior of components precisely and to focus on the what. Two examples of semantic abstracts are *signatures*, which describe the type information of a component, and *specifications*, which describe the dynamic behavior of a component. Signatures are really just a weak form of specification. Both are natural ways of describing components and have well-understood relations between instances of the abstract (e.g., type equivalence, logical relations between formal specifications), which we exploit heavily in defining matches.

We are interested in both signatures and specifications because they provide a range of expressiveness and "cost". On the one hand, signatures are "cheap" – for an existing component, the type is either required by or inferred by the compiler, and queries are easy for a programmer to write, since he or she is already familiar with the language's type system. On the other hand, although specifications require additional work, their expressive power is much greater.

The second factor in component matching is the granularity of the components, as illustrated by the y-axis of Figure 1.1. Components vary in size from individual language constructs to moderately-sized blocks of code to large software systems. In order to describe and reuse components, though, the components must be encapsulated in some way (e.g., function definitions, modular collections of functions, or stand-alone software systems). The granularities of software components in which we are interested are *functions* (e.g., C routines, Ada procedures, ML functions) and *modules* (e.g., C++ classes, Ada packages, ML modules). We are interested in both levels of match because in practice we expect users to want to reuse components at both levels of granularity.

The z-axis of Figure 1.1 represents the degree of relaxation of a match. It is rarely the case that we would require one component to match the other "exactly." In retrieval, we want a close match; as in any information retrieval context [Cor95, ML94, SM83], we might be willing to sacrifice precision for recall. That is, we would be willing to get some false positives as long as we do not miss any (or too many) true positives. In indexing, we use a partial ordering over a set of components, rather than equivalence between components. And in determining substitutability, we do not need the substituting component to have the exact same behavior as the substituted, only the same behavior relative to the environment that contains it. Therefore for each kind of match, we define both an *exact match* and various notions of *relaxed match*.

For example, relaxed signature matching on functions might allow reordering of a function's input parameters.

1.2.1 Component Signatures and Specifications

To be concrete in our examples and implementation, we have chosen particular languages for our signatures and specifications. We use ML [MTH90] as our component language, and hence rely on the ML type and module systems. Figure 1.2 shows three ML signature modules, *List*, *Queue*, and *Set*. (ML *signature* modules are akin to Ada *definition* modules and Modula-3 *interface* modules; ML implementations are written in modules called *structures*.) Each module signature contains a set of function signatures. For example, the *Queue* module contains four function signatures, including the function *deq* with signature $\alpha\ T \rightarrow \alpha$. Functions in a module are sometimes named with the module name as a prefix (e.g., *Queue.deq*). We explain the function signature notation in detail in Chapter 2; *deq*'s signature indicates that *deq* takes a queue of objects of some type and returns an object of that type.

signature $List$ =
  sig
    val $empty : unit \rightarrow \alpha\ list$
    val $cons : \alpha * \alpha\ list \rightarrow \alpha\ list$
    val $hd : \alpha\ list \rightarrow \alpha$
    val $tl : \alpha\ list \rightarrow \alpha\ list$
    val $map : (\alpha \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta\ list$
    val $intsort : (int * int \rightarrow bool) \rightarrow int\ \ list \rightarrow int\ \ list$
  end

signature $Queue$ =
  sig
    type $\alpha\ T$
    val $create : unit \rightarrow \alpha\ T$
    val $enq : \alpha\ T * \alpha \rightarrow \alpha\ T$
    val $rest : \alpha\ T \rightarrow \alpha\ T$
    val $deq : \alpha\ T \rightarrow \alpha$
  end

signature Set =
  sig
    type $\alpha\ T$
    val $create : unit \rightarrow \alpha\ T$
    val $insert : \alpha \rightarrow \alpha\ T \rightarrow \alpha\ T$
    val $delete : \alpha \rightarrow \alpha\ T \rightarrow \alpha\ T$
    val $member : \alpha \rightarrow \alpha\ T \rightarrow bool$
    val $union : \alpha\ T \rightarrow \alpha\ T \rightarrow \alpha\ T$
    val $intersection : \alpha\ T \rightarrow \alpha\ T \rightarrow \alpha\ T$
    val $difference : \alpha\ T \rightarrow \alpha\ T \rightarrow \alpha\ T$
  end

Figure 1.2: The Toy Signature Library (ML signature modules)

signature Queue = **sig**
  (*+ **using** Container +*)
  type $\alpha\ T$ (*+ **based on**
    Container.E Container.C +*)

  val $create : unit \rightarrow \alpha\ T$
  (*+ $create\ (\,) = q$
    ensures $q = empty$ +*)

  val $enq : \alpha\ T * \alpha \rightarrow \alpha\ T$
  (*+ $enq\ (q, e) = q2$
    ensures $q2 = insert\,(e, q)$ +*)

  val $rest : \alpha\ T \rightarrow \alpha\ T$
  (*+ $rest\ q = q2$
    requires $not\,(isEmpty\,(q))$
    ensures $q2 = butFirst\,(q)$ +*)

  val $deq : \alpha\ T \rightarrow \alpha$
  (*+ $deq\ q = e$
    requires $not\,(isEmpty\,(q))$
    ensures $e = first\,(q)$ +*)
end

Figure 1.3: A Larch/ML module specification

The expressiveness of a type system (and thus the effectiveness of signature matching) varies greatly across different programming languages. In a language such as C [KR78], functions operate on a few built-in base types (e.g., *int* or *double*) or pointers to them (e.g., $char\ *$) and thus, types are of limited expressiveness. In contrast, more advanced programming languages have rich type systems with user-defined abstract types, functional types, and polymorphic types, and thus types can convey more information about a component's behavior [Wad89]. For example, an ML function with signature $\alpha\ list \rightarrow \alpha$ takes as input a list of objects of some type α and returns an object of that type. Call the input list l and the returned object x. Because there is no way to generate objects of type α any other way, x must be an element of l (e.g., the first element, or a randomly-selected element). Thus, particularly for rich type systems, signatures provide a great deal of expressiveness.

Most type systems go only so far, however, in characterizing a component's behavior. For example, exactly *which* element of the list does the function in the example return? Specifications allow us to go further. We use Larch/ML [WRZ93] as our specification language. Larch/ML is a Larch interface language for ML, which we describe in more detail in Section 3.1. Figure 1.3 shows the Larch/ML module specification for *Queue*, which contains four function specifications. A function specification consists of a pre-condition clause (**requires**) and a post-condition clause (**ensures**), where each clause is an assertion in predicate logic. The interpretation of a function's specification is that the pre-condition implies the post-condition. For example, the function specification for *deq* specifies that if the input queue q is not empty, then the result returned by the function, e, is the first element of q. The operator *first* is defined in the *Container* trait (referenced in the **using** clause).

Specifications enable us to express very precise and detailed relationships between the behaviors of two components. For example, the C library routines *strcpy* and *strcat* have the same signature but we would be unhappy if one were substituted for the other. If we had specified the behavior we desired from a string function, we could compare that specification with the specifications of *strcpy* and *strcat* to see if either of those functions behaves as we desire.

Formal specifications may or may not be available for each component. Our hope is also that as more applications such as ours come to expect specifications, there will be more incentive for programmers to provide them.

1.2.2 Defining Match

Now we can instantiate component match (M in Declaration 1.2.1) for the four cases where components are either functions or modules and components are described by abstracts that are either signatures or specifications. Figure 1.4 shows how we partition the matches and in which chapter we define a particular class of matching. For functions, we consider signature and specification matches individually (Chapters 2 and 3, respectively). For modules, we define the matches independently of whether the abstracts are signatures or specifications (Chapter 4).

The first class of matching we consider is function signature matching. The various relaxed matches allow reordering of elements in a tuple, uncurrying of arguments to a function, renaming of type constructors, and instantiation of type variables. Allowing instantiation of type variables introduces both complexity and a great deal of expressiveness to the matches. We express each of the function signature match definitions in terms of whether we can find transformations to apply to the two function signatures such that the results are equal. Using transformations allows us to define match composition easily and cleanly. Chapter 2 describes function signature matching in detail.

The second class of matching is function specification matching. We define each of the matches in terms of a logical relationship, e.g., implication, between two specifications or between parts of the specifications. We relate all of the matches in a lattice. Chapter 3 describes function specification matching in detail.

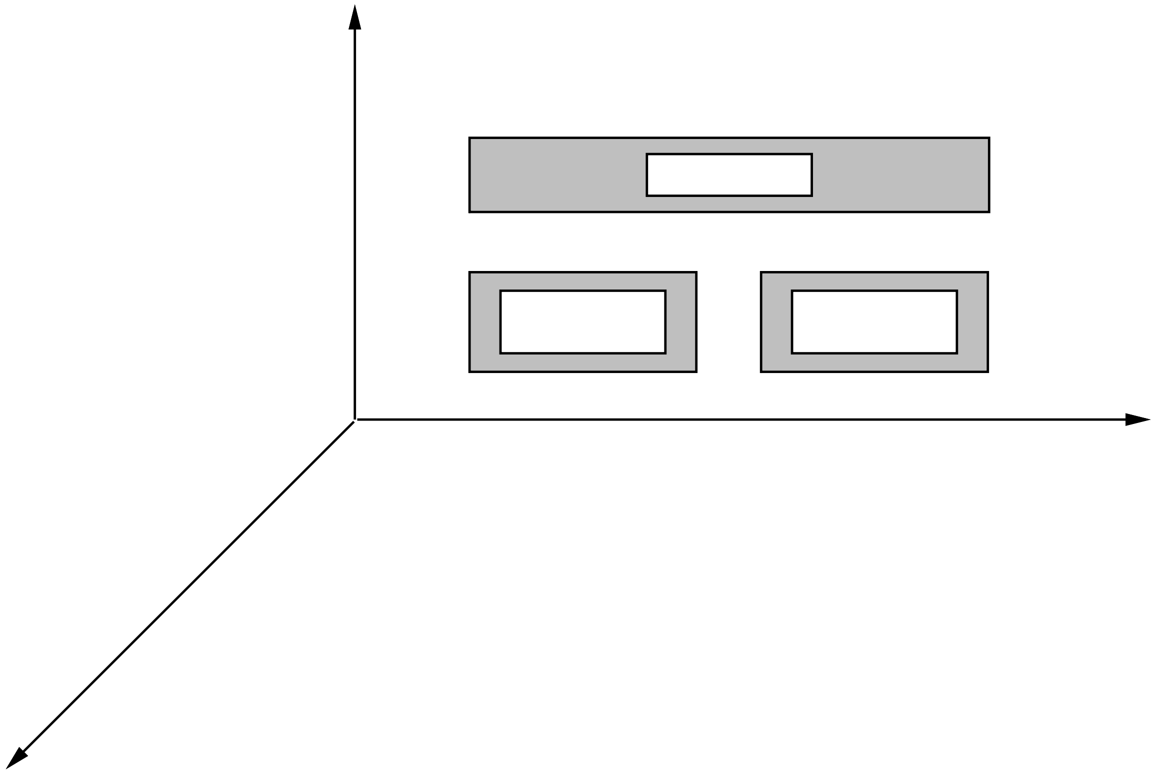


Figure 1.4: Which chapters define what

The third class of matching is module matching (both signature and specification). A module consists of some global information (e.g., type declarations) and a set of functions. Each module match requires some kind of correspondence between functions. The module match definitions are parameterized over what function match is used to determine this correspondence, which can be instantiated with any of the function matches in Chapters 2 or 3 (i.e., with either function signature or function specification match). Chapter 4 describes module matching in detail.

Our match definitions are orthogonal in several ways. Module match is parameterized by a function match that can be instantiated by *any* function match (either signature or specification). Function signature matches themselves are defined in such a way that the relaxed matches are composable. We could even consider signature match as a parameter to specification match.

Additionally, our general approach is flexible and extensible. The basic function signature match definitions apply to any statically-typed programming language (although some relaxed matches only apply if the language has particular features). Function specification matching is not even tied to formal specifications; the match definitions still apply for informal specifications, although proving a match must then be done informally. Moreover, we could use the same basic approach (identify the form of the abstract and define exact and various relaxed matches) to define matching for other kinds of abstracts, and then use those matches in the same way we use our signature and specification matches. An example of a different abstract is one that takes a keyword-based approach to specifications, so that a specification is a set of attribute-value pairs [PD89]. Another example is a specification abstract that also includes a *protocol* [AG94], which is a description of how the component expects to communicate with other components (e.g., remote procedure call in a client-server system architecture). Having protocols in addition to our existing specifications would allow us to detect mismatches in the way that two components communicate.

1.2.3 Applications

Recall the list of questions at the beginning of Section 1.1. Now we can ask the same questions using the concepts of function, module, signature, and specification:

1′. *Retrieval.* I need a function that returns an element from a given list of elements (i.e., a function with the type $\alpha\ list \rightarrow \alpha$).

2′. *Indexing.* What is the object class hierarchy that results from indexing the library?

3′. *Navigation.* Let me see all the nodes that are more general than $int * int \rightarrow bool$.

4′. *Substitution.* Is *Queue.deq* behaviorally equivalent to *Stack.pop*?

5′. *Subtyping.* Is *Stack* a subtype of *Bag*?

6′. *Modification.* Is there a reordering of arguments and an instantiation of variables such that the *create* function in the library (with type $int * \alpha \rightarrow \alpha\ list$) can be called instead of a function with type $bool * int \rightarrow bool\ list$?

We use the match predicates in three ways to answer these questions: (1) to compare a given component against all components in a set, (2) to compare components in a set pairwise, and (3) to compare two components directly. Formal descriptions of each of these classes of applications appear in Sections 5.1, 5.2, and 5.3, respectively. Each class of applications is defined in terms of the general component match M (Declaration 1.2.1), which can be instantiated by any of our match definitions.

The first class of applications uses match predicates to retrieve a subset of components in a library. Suppose we want to find all the components in a library, L, that are like a query component, Q. We can select an appropriate signature or specification match, M, and check $M(S, Q)$ for each $S \in L$. Retrieval itself has several kinds of applications. We can use retrieval to locate components for reuse (e.g., question 1′) and to analyze or to browse the library.

The second class of applications uses match predicates to build a hierarchical index on a library of components. An *indexed library* is useful for efficient storage and retrieval of components and for browsing. Using a subtype specification match, we could build an index for an object library to represent the subtype hierarchy (question $2'$). And while question $3'$ might at first glance look like a retrieval problem, if we are starting from the type of a component in the library, we can use the index of the library to answer the question.

The third class of applications simply compares two components using one of the matches. Questions $4'$, $5'$, and $6'$ are examples of applications in this class. Depending on the match, we get guarantees about whether various properties will hold if we substitute one component for the other. For example, if two function components have the same signature, then we can replace one with another and still be assured that our code will type check. In cases where a match is not exact, we may be able to use the information about how they are different to know what we need to change in order to reuse a component. For example, question $6'$ is answered by a function signature match that includes relaxations to allow reordering of arguments and instantiation of variables.

Specification matches define even stronger relationships between components. We may consider two components to be *behaviorally equivalent* if their pre-conditions are equivalent and their post-conditions are equivalent. We define this "exact pre/post match" in Section 3.2.1 and use this match to answer questions like $4'$. In Section 5.3.2, we use module match to define three different versions of subtyping, any of which we can use to answer questions like $5'$. Heuristic and text-based approaches to matching cannot answer questions like $4'$, $5'$, and $6'$ definitively, since the matches are not based on a formal and complete relationship between components.

Libraries

Since the second and third classes of applications assume a library of components, we briefly discuss libraries here, and explain the particular libraries we use in the thesis.

A *component library* is a set of components, either all functions or all modules. Given a library of modules, we form a library of functions by taking the union of the functions in each module. Another source of a function library is the set of built-in functions for a language.

In this thesis, we use three example libraries, summarized in Table 1.1. There are two small libraries, the *Toy Signature Library* and the *Toy Specification Library*. We use these to give examples of the various match definitions; they are very small so that we can easily see exactly which functions or modules are and are not matched by a query component. We use the *Community Library* to illustrate the results of our implementation of function signature matching on a moderate-sized library; all the examples in Section 5.1 and Section 5.2 use this library. The Community Library is built from three sub-libraries: the Edinburgh Library [Ber91], the SML/NJ Library [ATT93], and a CMU Library of local contributions [TR93].

Name	Figure	Sig or Spec	# of Modules	# of Functions
Toy Signature	Figure 1.2	Signature	3	17
Toy Specification	Figure 3.1	Specification	2	8
Community	–	Signature	129	1451
Edinburgh			45	401
SML/NJ			31	688
CMU			53	362

Table 1.1: Summary of libraries used in the thesis

1.3 Thesis Contributions

This dissertation makes the following contributions:

- *Foundational.* Within a general, highly modular, and extensible framework, we define matching for two kinds of semantic information (signatures and specifications) and two granularities of components (functions and modules). Each kind of matching has a generic form, within which all of the matches are related and may, in some cases, be composed. The orthogonality of the matches allows us to define match on modules independently of the particular match used on functions in the modules.
- *Applications.* We show how the definitions of matching can be applied to the problems of retrieval from libraries, indexing libraries, and reuse of components. We demonstrate the various signature and specification matches with examples of typical uses in each application.
- *Engineering.* We describe our implementations of function and module signature match, function specification match, function signature-based indexing, and function signature-based retrieval. These implementations demonstrate the feasibility of our approach and allow us to illustrate the applications with results from a moderately-sized component library.

1.4 Roadmap and Terminology

The remainder of the thesis is structured as follows. Chapters 2, 3, and 4 define signature and specification matching, as shown in Figure 1.4. In each chapter, we present notions of both exact and relaxed match, show how the definitions are related to each other, discuss our implementation of the matches, and evaluate the approach. Chapter 5 describes applications of the match definitions in the areas of retrieval (5.1), indexing (5.2), and substitution (5.3). We discuss related work in Chapter 6 and discuss conclusions and directions for future work in Chapter 7.

Throughout the definition chapters (Chapters 2 – 4), we give examples of matches for each definition. We give additional examples of the applications of signature and specification matching in Chapter 5. For each match, there is both a match name and a match predicate symbol. For example, the match predicate for function signature equivalence is named *exact match* (or *exact function signature match* when we are not clearly talking about function signatures) and has the predicate symbol $match_E$. For each match named M with the predicate symbol $match_M$ and components S and Q, if $match_M(S, Q)$, we say equivalently:

- S matches with Q (under M)
- M match of S with Q
- Q is matched by S (under M)
- Q retrieves S (under M)

It is important to distinguish between "matches with" and "is matched by", because not all matches are symmetric: $match_M(S, Q)$ does not necessarily imply that $match_M(Q, S)$. For the matches that are symmetric, we also say that "S and Q satisfy the match."

Chapter 2

Function Signature Matching

In this chapter, we examine various definitions of function signature matching. We begin in Section 2.1 with a description of what we mean by function signatures, present our definitions of the various matches in Section 2.2, and show how to compose definitions in Section 2.3. In Section 2.4, we define various properties of the matches, and show how each of the exact and relaxed matches are instances of a generic function signature match definition. We describe an implementation of the matches in Section 2.5.

2.1 Signatures

Function matching based on just signature information boils down to type matching, in particular matching *function types*. The following definition of types is based on Field and Harrison [FH88]. A *type* is either a *type variable* $\in$ *TypeVar* (denoted by Greek letters) or a *type constructor* $\in$ *TyCon* applied to other types. *Polymorphic* types contain at least one type variable; types that do not contain any type variables are *monomorphic*.

Each type constructor has an *arity* indicating the number of type arguments. *Base types* are constructors of 0-arity, e.g., *int*, *bool*; the "arrow" constructor for *function types* is binary, e.g., $int \rightarrow bool$. We use infix notation for tuple construction ($*$) and functions ($\rightarrow$), and otherwise use postfix notation for type constructors (e.g., *int list* stands for the "list of integers" type). The user-defined type, $\alpha\ T$, represents a type constructor T with arity 1, where the type of the argument to T is the type variable α.[1]

We assume that the type system includes tuples, polymorphism, higher-order functions, and user-defined types. These features are not necessary to do signature matching, but without them, some of the relaxed matches will not apply. We discuss this in more detail in Section 2.6.

[1]In ML, a common programming practice is to use T for the constructor name of the user-defined type of interest.

Definition 2.1.1 (Type Equality ($=_T$))

$\tau =_T \tau'$ iff
(1) they are lexically identical type variables or

(2) $\tau = tyCon(\tau_1, ..., \tau_n)$, $\tau' = tyCon'(\tau'_1, ..., \tau'_n)$,
$tyCon = tyCon'$, and $\forall\ 1 \leq i \leq n, \tau_i =_T \tau'_i$.

Variable Substitution

To allow substitution of other types for type variables, we introduce notation for *variable substitution*: $[\tau'/\alpha]\tau$ represents the type that results from replacing all occurrences of the type variable α in τ with τ', provided no variables in τ' occur in τ (read as "τ' replaces α in τ"). For example, $[(int \rightarrow int)/\beta](\alpha \rightarrow \beta) = \alpha \rightarrow (int \rightarrow int)$. A sequence of substitutions is right associative. For example, $[\beta/\gamma][\alpha/\beta](\beta \rightarrow \gamma) = [\beta/\gamma](\alpha \rightarrow \gamma) = (\alpha \rightarrow \beta)$. In a case like the previous example, where τ' is just a variable, $[\tau'/\alpha]\tau$ is simply *variable renaming.* The concatenation of two sequences is denoted with a "$\hat{}$"; $U1\,\hat{}\,U2\tau = U1(U2\tau)$.

We use the same notation for renaming of type constructors. In this case, $[c'/c]\tau$ (where $c, c' \in TyCon$) represents the type that results from replacing all occurrences of the type constructor c in τ with c', provided c' does not occur in τ. For example, $[Set/T](\alpha\ T \rightarrow \alpha) = \alpha\ Set \rightarrow \alpha$. The $\rightarrow$ and $*$ type constructors cannot be renamed.

We will use V for a sequence of variable renamings, V_{TC} for a sequence of type constructor renamings, and U for a sequence of variable substitutions.

2.2 Match Definitions

Given the type of a function from a component library, τ_l, and the type of a query, τ_q, we define a generic form of function signature match, $M(\tau_l, \tau_q)$, as follows:

Definition 2.2.1 (Generic Function Signature Match)
$M(\tau_l, \tau_q) = \exists$ a *transformation pair*, $T = (T_l, T_q)$, such that $T_l(\tau_l)\ \mathcal{R}\ \ T_q(\tau_q)$

where the implicit parameter $\mathcal{R}$ is some relationship between types (e.g., equality) and T_l and T_q are transformations that are applied to the library and query types, respectively. A transformation is a function from types to types (e.g., a function that reorders elements in a tuple). Most of the matches we define apply transformations to only one of the types. Where possible, we apply the transformation to the library type, τ_l, in which case T_q is simply the identity function. For example, in exact match, two types match if they are equal modulo variable renaming. In this case, T_l is a sequence of variable renamings, T_q is the identity function, and $\mathcal{R}$ is the type equality ($=_T$) relation.

We classify relaxed signature matches as either *partial* matches, which vary $\mathcal{R}$, the relationship between τ_l and τ_q (e.g., define $\mathcal{R}$ to be a partial order), or *transformation* matches, which vary T_l or T_q, the transformations on types. In the following sections, we first define exact match, followed by transformation matches, partial matches, and combined matches. We illustrate each definition with examples that use the definition to retrieve functions from the Toy Signature Library in Figure 1.2 (page 1.2).

2.2.1 Exact Match

Definition 2.2.2 (Exact Match)

$$\begin{aligned} match_E(\tau_l, \tau_q) \quad = \quad & \exists \text{ a sequence of variable renamings, } V \text{, such that} \\ & V \ \tau_l =_T \tau_q \end{aligned}$$

Two function types match exactly if they match modulo variable renaming. For monomorphic types, there are no variables, so $match_E(\tau_l, \tau_q) = (\tau_l =_T \tau_q)$ where τ_l and τ_q are monomorphic. We only need a sequence of renamings for one of the type expressions, since for any two renamings, V_1 and V_2 such that $V_1\tau_1 =_T V_2\tau_2$, we could construct a V' such that $V'\tau_1 =_T \tau_2$. (Note we could consider $match_E$ as a form of transformation match since it allows variable renaming.)

For polymorphic types, actual variable names do not matter, provided there is a way to rename variables so that the two types are identical. For example, $\tau_l = (\alpha * \alpha) \rightarrow bool$ matches with $\tau_q = (\beta * \beta) \rightarrow bool$ with the substitution $V = [\beta/\alpha]$. But $\tau_l = \alpha \rightarrow \beta$ does not match with $\tau_q = \gamma \rightarrow \gamma$ because once we substitute γ for α to get $\gamma \rightarrow \beta$, we cannot substitute γ for β, since γ already occurs in the type. This is the "right thing" because the difference between τ_l and τ_q is more than just variable names; τ_q takes a value of some type γ and returns a value of *the same* type, whereas τ_l takes a value of some type and returns a value of a potentially *different* type.

To see how exact match might be used in practice, suppose a user wants to locate a function in the Toy Signature Library that applies an input function to each element of a list, forming a new list. The query $\tau_q = (\alpha \rightarrow \gamma) \rightarrow \alpha\ list \rightarrow \gamma\ list$ is matched by the *map* function (with the renaming $[\gamma/\beta]$), exactly what the user wants.

2.2.2 Transformation Relaxations

Exact match is a useful starting point, but it may miss useful functions whose types are close but do not exactly match the query. Exact match requires a user to be either familiar with a library or lucky in choosing the exact syntactic format of a type.

One class of relaxed match *transforms* a type expression to achieve a match. Examples include renaming type constructors, changing whether a function is curried or uncurried, changing the order of types in a tuple, and changing the order of arguments to a function (for functions that take more than one argument). These last two are similar since we can view multiple arguments to a function as a tuple. The following two queries illustrate the need for transformation relaxations. The query $\tau_q = \alpha \rightarrow \alpha\ list \rightarrow \alpha\ list$ would miss the *cons* function because τ_q is curried while *cons* is not. The query $\tau_q = (\alpha\ list * \alpha) \rightarrow \alpha\ list$ would miss *cons* because the types in the tuple are in a different order.

Type Constructor Renaming

Most type systems have a small set of built-in type constructors (e.g., *list*) but allow users to add new types using user-defined type constructors (e.g., the *Queue* and *Set* modules in the Toy Signature Library both have the user-defined type constructor T). In the same way that queriers should not have to guess a type variable name, neither should they have to guess a type constructor name defined by someone else, since different users may use a different name for the same type constructor. *Type constructor match* allows renaming of type constructors for these cases. We do not include this renaming in the exact match, since there may also be cases where a querier does want to restrict the matches to those with exactly the same type constructors. We could choose to restrict renaming to user-defined types, but that would make an unnecessary distinction between a built-in type like *list* and a user-defined type, since for a query like $\alpha\ T \rightarrow \alpha$, we would want T to match not just user-defined type constructors, but also *list*. We exclude the type constructors $\rightarrow$ and $*$ from renaming, since they have special meanings in the type system and we expect users to be familiar with them (and hence not to need to rename them).

Definition 2.2.3 (Type Constructor Match)

$$\begin{aligned} match_{tycon}(\tau_l, \tau_q) \quad = \quad & \exists \text{ a sequence of type constructor renamings, } V_{TC}, \text{ such that} \\ & match_E(V_{TC}\ \tau_l, \tau_q) \end{aligned}$$

As an example, suppose a user wants a function to return the first element of a list with the query $\alpha\ C \rightarrow \alpha$. Under exact match, this query is not matched by any functions in the Toy Signature Library, but under type constructor match, the query retrieves the functions *hd* on lists (with renaming $[C/list]$) and *deq* on queues (with renaming $[C/T]$).

As another example, suppose a user wants to locate a function to add an element to a collection with the query $\tau_q = (\alpha\ C * \alpha) \rightarrow \alpha\ C$. Under type constructor match, this query retrieves the *enq* function on queues (with the renaming $[C/T]$), which may be what the user wants. However, the query is not matched by the *cons* function on lists or the *insert* and *delete* functions on sets, other likely candidates.

Uncurrying Functions

A function that takes multiple arguments may be either curried or uncurried. The uncurried version of a function has a type $(\tau_1 * \ldots * \tau_{n-1}) \rightarrow \tau_n$, while the corresponding curried version has a type $\tau_1 \rightarrow \ldots \rightarrow \tau_{n-1} \rightarrow \tau_n$. In many cases, it will not matter to the querier whether or not a function is curried. We define uncurry match by applying the uncurry transformation to both query and library types. We choose to uncurry rather than curry each type so that we can later compose this relaxed match with one that reorders the types in a tuple.

The *uncurry transformation*, UC, produces an uncurried version of a given type:

$$UC(\tau) = \begin{cases} (\tau_1 * \ldots * \tau_{n-1}) \rightarrow \tau_n & \text{if } \tau = \tau_1 \rightarrow \ldots \rightarrow \tau_{n-1} \rightarrow \tau_n, n > 2 \\ \tau & \text{otherwise} \end{cases}$$

The uncurry transformation is non-recursive; any nested functions will not be uncurried. We also define a recursive version, UC^+:

$$UC^+(\tau) = \begin{cases} (UC^+(\tau_1) * \ldots * UC^+(\tau_{n-1})) \rightarrow UC^+(\tau_n) & \text{if } \tau = \tau_1 \rightarrow \ldots \rightarrow \tau_{n-1} \rightarrow \tau_n, n > 2 \\ tyCon(UC^+(\tau_1), \ldots, UC^+(\tau_n)) & \text{if } \tau = tyCon(\tau_1, \ldots, \tau_n) \\ \tau & \text{where } \tau \text{ is a variable or a base type} \end{cases}$$

For example, if $\tau = int \rightarrow int \rightarrow (int \rightarrow int \rightarrow bool) \rightarrow bool$ then $UC(\tau) = (int * int * (int \rightarrow int \rightarrow bool)) \rightarrow bool$ and $UC^+(\tau) = (int * int * ((int * int) \rightarrow bool)) \rightarrow bool$.

Definition 2.2.4 (Uncurry Match and Recursive Uncurry Match)

$$match_{uncurry}(\tau_l, \tau_q) = match_E(UC(\tau_l), UC(\tau_q))$$

$$match_{uncurry^+}(\tau_l, \tau_q) = match_E(UC^+(\tau_l), UC^+(\tau_q))$$

Uncurry match takes two uncurried function types and determines whether their corresponding argument types match. *Recursive uncurry match* is similar but allows recursive uncurrying of τ_l's and τ_q's functional arguments. By applying the UC (or UC^+) transformation to both τ_l and τ_q, we are transforming the types into a canonical form, and then checking that the resulting types are equal (modulo variable renaming).

Suppose we again are looking for a function that adds an element to a collection. But this time suppose we use the query $\tau_q = \alpha\ T \rightarrow \alpha \rightarrow \alpha\ T$. Exact match yields nothing, but uncurry match would return the function *enq* on queues. Note that again this query does not retrieve *cons*, *insert*, or *delete*.

Since the *uncurry* transformation is applied to both the query and library types, it is not necessary to define an additional *curry match*. Such a match would be similar in structure, relying on a *curry transformation* to produce a curried version of a given type; that is,

$match_{curry}(\tau_l, \tau_q) = match_E(curry(\tau_l), curry(\tau_q))$. The curry and uncurry transformations are not exactly inverses, since it is not always true that $curry(UC(\tau)) = \tau$ or that $UC(curry(\tau)) = \tau$. However, the two matches define the same equivalence: $match_{curry}(\tau_l, \tau_q)$ if and only if $match_{uncurry}(\tau_l, \tau_q)$.

Reordering Tuples

Tuples group multiple arguments to a function, but sometimes the order of the arguments does not matter. For example, a function to test membership in a list could have type $(\alpha * \alpha\ list) \rightarrow bool$ or type $(\alpha\ list * \alpha) \rightarrow bool$. *Reorder match* allows matching on types that differ only in their order of arguments.

We define reorder match in terms of permutations. Given a function type whose first argument is a tuple (e.g., $\tau = (\tau_1 * \ldots * \tau_{n-1}) \rightarrow \tau_n$), a *reorder transformation*, T_σ, defines a *permutation* σ, which is applied to the tuple. σ is a bijection with domain and range $1 \ldots n-1$ such that $T_\sigma(\tau) = (\tau_{\sigma(1)} * \ldots * \tau_{\sigma(n-1)}) \rightarrow \tau_n$.

Definition 2.2.5 (Reorder Match)

$$\begin{aligned} match_{reorder}(\tau_l, \tau_q) \quad = \quad & \exists \text{ a reorder transformation } T_\sigma \text{ such that} \\ & match_E(T_\sigma(\tau_l), \tau_q) \end{aligned}$$

Under this relaxation, a library type, τ_l, matches with a query type, τ_q, if the argument types of τ_l can be reordered so that the types match exactly. Although we choose to apply the reorder transformation, T_σ, to the library type τ_l, we could equivalently apply the inverse, $T_{\sigma^{-1}}$, to the query type τ_q: $match_E(T_\sigma(\tau_l), \tau_q) = match_E(\tau_l, T_{\sigma^{-1}}(\tau_q))$. With reorder match, the query $\tau_q = (\alpha\ list * \alpha) \rightarrow \alpha\ list$ we discussed at the beginning of Section 2.2.2 is now matched by the desired list function, *cons*.

There are two variations on reorder match: we can allow (1) recursive permutations so that a tuple's component types may be reordered ($match_{reorder^+}$); and (2) reordering of arguments to user-defined type constructors, e.g., so that $(int, \alpha)\ T \rightarrow int$ and $(\alpha, int)\ T \rightarrow int$ would match.

2.2.3 Partial Relaxations

Often a user with a specific query type, e.g., $int\ list \rightarrow int\ list$, could just as easily use an instantiation of a more general function, e.g., $\alpha\ list \rightarrow \alpha\ list$. Or, the user may have difficulty determining the most general type of the desired function but can give an example of what is desired. Allowing more general types to match a query type accommodates these kinds of situations. Conversely, we can also imagine cases where a user asks for a general type that does not match anything in the library exactly. There may be a useful function in the library whose type is more specific, but the code could be easily generalized to be useful to the user. We define *generalized* and *specialized* match to address both of these cases.

Referring back to our definition of generic function signature match (Definition 2.2.1), for exact match, the relation, $\mathcal{R}$, between types is equality. For *partial* matches we relax this relation to be a partial order on types. We use variable substitution to define the partial ordering, based on the "generality" of the types. For example, $\alpha \rightarrow \alpha$ is a generalization of infinitely many types, including $int \rightarrow int$ and $(int * \beta) \rightarrow (int * \beta)$, using the variable substitutions $[int/\alpha]$ and $[(int * \beta)/\alpha]$, respectively.

τ is *more general than* τ' ($\tau \geq \tau'$) if the type τ' is the result of a (possibly empty) sequence of variable substitutions applied to type τ. Equivalently, we say τ' is an *instance* of τ ($\tau' \leq \tau$). We would typically expect functions in a library to have as general a type as possible.

Generalized Match

Definition 2.2.6 (Generalized Match)

$$match_{gen}(\tau_l, \tau_q) \quad = \quad \tau_l \geq \tau_q$$

A library type matches with a query type if the library type is more general than the query type. Exact match, with variable renaming, is really just a special case of generalized match where all the variable substitutions are variable renamings, so $match_E(\tau_l, \tau_q) \Rightarrow match_{gen}(\tau_l, \tau_q)$.

For example, suppose a user needs a function to convert a list of integers to a list of boolean values, where each boolean corresponds to whether or not the corresponding integer is positive. The user might write a query like $\tau_q = (int \rightarrow bool) \rightarrow int\ list \rightarrow bool\ list$. Under exact match, this query is not matched by any function in our library. But under generalized match, τ_q would retrieve *map*, since *map*'s type is more general than the query type. This kind of match is especially desirable, since the user does not need to make any changes to use the more general function.

Specialized Match

Definition 2.2.7 (Specialized Match)

$$match_{spcl}(\tau_l, \tau_q) \quad = \quad \tau_l \leq \tau_q$$

Specialized match is the inverse of generalized match; we could alternatively define $match_{spcl}$ in terms of $match_{gen}$ by swapping the order of the types: $match_{spcl}(\tau_l, \tau_q) = match_{gen}(\tau_q, \tau_l)$. It also follows that exact match is a special case of specialized match: $match_E(\tau_l, \tau_q) \Rightarrow match_{spcl}(\tau_l, \tau_q)$

As an example of how specialized match is useful, suppose the querier needs a general function to sort lists and uses the query $\tau_q = ((\alpha * \alpha) \rightarrow bool) \rightarrow \alpha\ list \rightarrow \alpha\ list$. Our library does not contain such a function, but under specialized match, τ_q would retrieve *intsort*, an integer sorting function with the type $\tau_l = ((int * int) \rightarrow bool) \rightarrow int\ list \rightarrow int\ list$. Assuming *intsort* is written reasonably well, it should be easy for the querier to modify it to sort arbitrary objects since the comparison function is passed as a parameter.

Specialized match is also useful in cases where we do not know an actual type for part of the query. For example, if we wanted a function to compare strings, the return value might be a boolean, or an integer (e.g., measuring edit distance), or an enumerated type. The query $string * string \rightarrow \alpha$ with specialized match would retrieve all such functions. Sections 5.1.2 and 5.1.3 contain many examples using specialized match for statistical analysis and for browsing on libraries.

Although we present generalized and specialized match in terms of changing the relation ($\mathcal{R}$) between τ_l and τ_q, we could also define them as transformation matches, since the definition of the $\leq$ relation on types is in terms of variable substitution.

Definition 2.2.8 ((alternative) Generalized and Specialized Match)

$$\begin{aligned} match_{gen}(\tau_l, \tau_q) &= \exists \text{ a sequence of variable substitutions, } U, \text{ such that} \\ &\quad match_E(U\ \tau_l, \tau_q) \\ \\ match_{spcl}(\tau_l, \tau_q) &= \exists \text{ a sequence of variable substitutions, } U, \text{ such that} \\ &\quad match_E(\tau_l, U\ \tau_q) \end{aligned}$$

We can even define $match_{gen}(\tau_l, \tau_q)$ as $U\tau_l =_T \tau_q$; the use of $match_E$ is redundant since generalized match requires a sequence of substitutions that includes any necessary variable renaming. We will appeal to the above alternative definitions of generalized and specialized match when we define the composition of different relaxed matches (Section 2.3).

2.3 Combining Relaxations

Each relaxed match is individually a useful match to apply when searching for a function of a given type. Combinations of these separately defined relaxed matches widen the set of library types retrieved. Suppose again that a user wants a function to add an element to a collection. This function might have one of four possible types:

1. $\alpha * \alpha\ T \rightarrow \alpha\ T$
2. $\alpha\ T * \alpha \rightarrow \alpha\ T$
3. $\alpha \rightarrow \alpha\ T \rightarrow \alpha\ T$
4. $\alpha\ T \rightarrow \alpha \rightarrow \alpha\ T$

Under reorder match, a query of type 1 or 2 retrieves library functions of types 1 or 2, but not types 3 or 4. Under uncurry match, a query of type 1 or 3 retrieves library functions of those types (and likewise for types 2 and 4). But no individual relaxed match allows a single query to retrieve all four types. By composing reorder and uncurry match, a query of any of the four types will retrieve library functions of all four types, which is what we would like.

We deliberately gave our definitions in a form so that we can easily compose them. If we use the alternative definitions of $match_{gen}$ and $match_{spcl}$, each of the relaxed match definitions presented in Sections 2.2.2 and 2.2.3 can be cast in a composable form by instantiating $\mathcal{R}$ to $match_E$ in the generic function match (Definition 2.2.1):

$$\exists \text{ a } \textit{transformation pair}, T = (T_l, T_q), \text{ such that } match_E(T_l(\tau_l), T_q(\tau_q)).$$

The *match composition* of two relaxed matches, denoted as $(match_{R1} \circ\ match_{R2})$, is defined by composing the transformations on each type, applying the inner (R2) relaxation first.

Definition 2.3.1 (Match Composition)

$$\begin{aligned}(match_{R1} \circ\ match_{R2})(\tau_l, \tau_q) \quad = \quad & \exists \text{ transformation pairs } T1 = (T1_l, T1_q) \text{ and } T2 = (T2_l, T2_q) \\ & \text{such that } match_E(T1_l \circ\ T2_l(\tau_l), T1_q \circ\ T2_q(\tau_q))\end{aligned}$$

The choice of $R1$ determines the kinds of transformations in $T1$ (and likewise for $R2$ and $T2$). For $match_{tycon}$, T_l is a sequence of type constructor renamings and T_q is the identity function. For $match_{uncurry}$, T_l and T_q are UC; the "$\exists$" is not necessary, since there is only one possible uncurry transformation. For $match_{reorder}$, T_l is a reorder transformation and T_q is the identity function. For $match_{gen}$, T_l is a sequence of variable substitutions and T_q is the identity function. For $match_{spcl}$, T_l is the identity function and T_q is a sequence of variable substitutions.

We can compose any number of relaxed matches in any order. The order in which they are composed does make a difference; match composition is not commutative, as we show in detail in Section 2.4.2. For simplicity, we omit the recursive versions of $match_{uncurry^+}$ and $match_{reorder^+}$, although the analysis below could be easily extended to include them. Thus, there are five "basic" relaxed matches: $match_{tycon}$, $match_{uncurry}$, $match_{reorder}$, $match_{gen}$, and $match_{spcl}$. That is, $R1, R2 \in$ {tycon, uncurry, reorder, gen, spcl}.

We now consider some of the interesting combinations of these relaxed matches. In cases where a pair of relaxations is not commutative, we order the relaxations in the way that allows the most matches. So, for example, uncurry and reorder match are not commutative, but $match_{uncurry} \circ\ match_{reorder}(\tau_l, \tau_q) \Rightarrow match_{reorder} \circ\ match_{uncurry}(\tau_l, \tau_q)$, so we use $match_{reorder} \circ\ match_{uncurry}(\tau_l, \tau_q)$.

- $(match_{reorder} \circ match_{uncurry})(\tau_l, \tau_q)$

  With this composition, two types match if they are equivalent modulo whether or not they are curried or whether or not the arguments are in the same order. We uncurry the types first, thereby allowing a reordering on any tuples formed by uncurrying. Using this composition, the query type $\tau_q = \alpha\ T \rightarrow \alpha \rightarrow \alpha\ T$ would be matched by *enq* ($\tau_l = \alpha\ T * \alpha \rightarrow \alpha\ T$) on queues and *insert* and *delete* ($\tau_l = \alpha \rightarrow \alpha\ T \rightarrow \alpha\ T$) on sets.

- $(match_{tycon} \circ match_{reorder} \circ match_{uncurry})(\tau_l, \tau_q)$

  τ_l and τ_q match if they are equivalent modulo whether or not they are curried, whether or not the arguments are in the same order, and with renaming of type constructors. For example, the query $\tau_q = \alpha\ T \rightarrow \alpha \rightarrow \alpha\ T$ would be matched by *cons* ($\tau_l = \alpha * \alpha\ list \rightarrow \alpha\ list$) as well as *enq*, *insert*, and *delete*.

- $(match_{uncurry} \circ match_{gen})(\tau_l, \tau_q)$

  τ_l and τ_q match if the uncurried form of the result of applying a sequence of variable substitutions to τ_l is equivalent to the uncurried form of τ_q. With this composition, the query $\tau_q = ((int \rightarrow bool) * int\ list) \rightarrow bool\ list$ would be matched by the *map* function ($\tau_l = (\alpha \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta\ list$).

- $(match_{reorder} \circ match_{uncurry} \circ match_{gen})(\tau_l, \tau_q)$

  τ_l and τ_q match if some permutation of the uncurried form of τ_l' is equivalent to the uncurried form of τ_q, where τ_l' is the result of applying a sequence of variable substitutions to τ_l. Using this combined match, the query $\tau_q = (int\ list, (int \rightarrow bool)) \rightarrow bool\ list$ is matched by the *map* function in our library ($\tau_l = (\alpha \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta\ list$).

- $(match_{tycon} \circ match_{reorder} \circ match_{uncurry} \circ match_{gen})(\tau_l, \tau_q)$

  This is the same as the previous composition except it also allows renaming of type constructors. Under this match, $\tau_q = int\ C \rightarrow int \rightarrow int\ C$ retrieves *cons*, *enq*, *insert*, and *delete*.

- $(match_{gen} \circ match_{spcl})(\tau_l, \tau_q)$

  τ_l and τ_q match if the result of applying a sequence of variable substitutions, $U1$, to τ_l is equivalent to the result of applying a sequence of variable substitutions, $U2$, to τ_q. Note that there is no constrain on any relationship between $U1$ and $U2$. Thus, this composed match is not equivalent to type unification. For example, under this composed match, the type $\tau_l = bool \rightarrow \alpha$ matches with the query $\tau_q = \alpha \rightarrow int$ with substitutions $U1 = [int/\alpha]$ and $U2 = [bool/\alpha]$. But τ_l and τ_q are not unifiable, since unification applies the same renaming to both types and so α would have to be renamed consistently.

2.4 Properties of the Matches

2.4.1 Equivalence and Partial Order

The function signature match definitions are relations on types. Thus, we can classify the matches according to whether they are equivalences, partial orders, or neither. We use this classification to build indexed libraries (Section 5.2).

Definition 2.4.1 (Equivalence Match)

A match relation $M_=$ is an *equivalence match* if:

1. $M_=(\tau, \tau)$ for all types τ [Reflexive]
2. If $M_=(\tau_1, \tau_2)$ then $M_=(\tau_2, \tau_1)$ [Symmetric]
3. If $M_=(\tau_1, \tau_2)$ and $M_=(\tau_2, \tau_3)$ then $M_=(\tau_1, \tau_3)$ [Transitive]

Definition 2.4.2 (Partial Order Match)

A match relation $M_\geq$ is a *partial order match* if:

1. $M_\geq(\tau, \tau)$ for all types τ [Reflexive]
2. If $M_\geq(\tau_1, \tau_2)$ and $M_\geq(\tau_2, \tau_1)$ then $M_=(\tau_1, \tau_2)$ [Antisymmetric]
3. If $M_\geq(\tau_1, \tau_2)$ and $M_\geq(\tau_2, \tau_3)$ then $M_\geq(\tau_1, \tau_3)$ [Transitive]

Equivalence matches partition types into sets of types that are equivalent modulo some transformations. Type equivalence ($=_T$) and most of the match definitions in this chapter (exact, type constructor, uncurry, and reorder matches) are equivalence matches. Partial order matches impose an ordering on the types. To show the antisymmetric property of a partial order match, we need a *corresponding equivalence match*, $M_=$. Generalized and specialized match are partial order matches with $M_= = match_E$. Table 2.1 summarizes this classification, as well as showing the predicate symbols for each match.

Proving the properties of each of the match definitions is fairly straightforward. Most of the proofs about these matches and their compositions rely on properties of the underlying transformations on types. Consider the case for showing that exact match is an equivalence match.

1. Exact match is reflexive ($match_E(\tau, \tau)$), since for any type τ, $\tau =_T \tau$ with no variable renaming.

2. Exact match is symmetric. Suppose $match_E(\tau_1, \tau_2)$. Then (by definition of exact match) there is a sequence of variable renamings V such that $V\tau_1 =_T \tau_2$. For a sequence of variable renamings $V = [u_1/v_1] \ldots [u_n/v_n]$, define the inverse, $V^{-1} = [v_n/u_n] \ldots [v_1/u_1]$. We can prove by induction on the length of V that if $V\tau_1$ is "valid" (i.e., the occurs requirement is satisfied for each renaming), and $V\tau_1 =_T \tau_2$, then (1) $V^{-1}\tau_2$ is valid, and (2) $V^{-1}\tau_2 =_T \tau_1$. Thus, $match_E(\tau_2, \tau_1)$ using variable renaming V^{-1}.

Name of Match	Predicate Symbol	Kind of Match
Exact	$match_E$	Equivalence
Type Constructor	$match_{tycon}$	Equivalence
Uncurry	$match_{uncurry}$	Equivalence
Reorder	$match_{reorder}$	Equivalence
Generalized	$match_{gen}$	Partial Order
Specialized	$match_{spcl}$	Partial Order

Table 2.1: Function signature matches, their symbols and classifications

3. Exact match is transitive. Suppose $match_E(\tau_1, \tau_2)$ and $match_E(\tau_2, \tau_3)$. Then there exist variable renaming sequences $V1$ and $V2$ such that $V1\ \tau_1 =_T \tau_2$ and $V2\ \tau_2 =_T \tau_3$. Let $V3 = V2 \hat{} V1$. Then $V3\ \tau_1 = V2(V1\ \tau_1) =_T V2\ \tau_2 =_T \tau_3$, so $match_E(\tau_1, \tau_3)$.

For generalized match, the proofs for reflexivity and transitivity are the same as for exact match. However, because the transformations are variable substitutions rather than variable renamings, we cannot guarantee the existence of an inverse substitution sequence, and hence generalized match is not symmetric. Further, suppose $match_{gen}(\tau_1, \tau_2)$ and $match_{gen}(\tau_2, \tau_1)$. Then there exist variable substitution sequences $U1$ and $U2$ such that $U1\ \tau_1 =_T \tau_2$ and $U2\ \tau_2 =_T \tau_1$. We can prove from this that $U1$ and $U2$ must be variable renamings, and hence $match_E(\tau_1, \tau_2)$, and generalized match is antisymmetric. Using the antisymmetry property of generalized match, and the definition of specialized match as the inverse of generalized match, we can prove that two types match under both generalized and specialized match if and only if the two types are equivalent.

Lemma 2.4.1 $match_{gen}(\tau_l, \tau_q) \wedge match_{spcl}(\tau_l, \tau_q) \Leftrightarrow match_E(\tau_l, \tau_q)$

Proof

$\Rightarrow$: Assume $match_{gen}(\tau_l, \tau_q) \wedge match_{spcl}(\tau_l, \tau_q)$. Since $match_{gen}$ and $match_{spcl}$ are inverses, $match_{spcl}(\tau_l, \tau_q) = match_{gen}(\tau_q, \tau_l)$. It follows from antisymmetry of $match_{gen}$ that $match_E(\tau_l, \tau_q)$.

$\Leftarrow$: Assume $match_E(\tau_l, \tau_q)$. Let $U1 = U2 = Id$ (i.e., the identity function). Then $match_E(U1\tau_l, \tau_q)$ and $match_E(\tau_l, U2\tau_q)$, so $match_{gen}(\tau_l, \tau_q)$ and $match_{spcl}(\tau_l, \tau_q)$ □

2.4.2 Match Composition

Commutativity

Not all match compositions are commutative. In particular, uncurry match does not commute with either reorder, generalized, or specialized match; additionally, reorder and generalized match do not commute. We list pairs of relaxations whose composition is commutative in Theorem 2.4.2. For pairs that are not commutative, we show which order of composition is stronger in Theorem 2.4.3.

Theorem 2.4.2 *The following match compositions are commutative:*

1. $match_{reorder} \circ match_{spcl}(\tau_l, \tau_q) = match_{spcl} \circ match_{reorder}(\tau_l, \tau_q)$
2. $match_{gen} \circ match_{spcl}(\tau_l, \tau_q) = match_{spcl} \circ match_{gen}(\tau_l, \tau_q)$
3. $match_{tycon} \circ match_{spcl}(\tau_l, \tau_q) = match_{spcl} \circ match_{tycon}(\tau_l, \tau_q)$
4. $match_{tycon} \circ match_{reorder}(\tau_l, \tau_q) = match_{reorder} \circ match_{tycon}(\tau_l, \tau_q)$
5. $match_{tycon} \circ match_{uncurry}(\tau_l, \tau_q) = match_{uncurry} \circ match_{tycon}(\tau_l, \tau_q)$
6. $match_{tycon} \circ match_{gen}(\tau_l, \tau_q) = match_{gen} \circ match_{tycon}(\tau_l, \tau_q)$

Proof Sketch

(1, 2, and 3): For specialized match, the transformation is applied to the second type, while for reorder, generalized, or type constructor match, the transformation is applied to the first type. The two transformations are independent, so the order in which the two transformations is applied does not matter:

$$match_{reorder} \circ match_{spcl}(\tau_l, \tau_q) = \exists\, T_\sigma, U : match_E(T_\sigma(\tau_l), U\tau_q) = match_{spcl} \circ match_{reorder}(\tau_l, \tau_q)$$
$$match_{gen} \circ match_{spcl}(\tau_l, \tau_q) = \exists\, U1, U2 : match_E(U1\tau_l, U2\tau_q) = match_{spcl} \circ match_{gen}(\tau_l, \tau_q)$$
$$match_{tycon} \circ match_{spcl}(\tau_l, \tau_q) = \exists\, V_{TC}, U : match_E(V_{TC}\ \tau_l, U\tau_q) = match_{spcl} \circ match_{tycon}(\tau_l, \tau_q)$$

(4) $match_{tycon} \circ match_{reorder}(\tau_l, \tau_q) = match_{reorder} \circ match_{tycon}(\tau_l, \tau_q)$:

(a) $match_{tycon} \circ match_{reorder}(\tau_l, \tau_q) \Rightarrow match_{reorder} \circ match_{tycon}(\tau_l, \tau_q)$

Assume $match_{tycon} \circ match_{reorder}(\tau_l, \tau_q)$. Then $\exists\, V_{TC}, T_\sigma : match_E(V_{TC} \circ T_\sigma(\tau_l), \tau_q)$. $T_\sigma \circ V_{TC}(\tau) = V_{TC} \circ T_\sigma(\tau)$ (by applying the definitions of V_{TC} and T_σ). So $match_E(T_\sigma \circ V_{TC}(\tau_l), V_{TC} \circ T_\sigma(\tau_l))$. Therefore, since $match_E$ is an equivalence match, $match_E(T_\sigma \circ V_{TC}(\tau_l), \tau_q)$, so $match_{reorder} \circ match_{tycon}(\tau_l, \tau_q)$.

(b) $match_{reorder} \circ match_{tycon}(\tau_l, \tau_q) \Rightarrow match_{tycon} \circ match_{reorder}(\tau_l, \tau_q)$

The proof is similar to (a).

(5) $match_{tycon} \circ match_{uncurry}(\tau_l, \tau_q) = match_{uncurry} \circ match_{tycon}(\tau_l, \tau_q)$:

The proof is similar to that for (4), using the fact that $V_{TC} \circ UC(\tau) = UC \circ V_{TC}(\tau)$, which follows from the definitions of V_{TC} and UC.

(6) $match_{tycon} \circ match_{gen}(\tau_l, \tau_q) = match_{gen} \circ match_{tycon}(\tau_l, \tau_q)$:

(a) $match_{tycon} \circ match_{gen}(\tau_l, \tau_q) \Rightarrow match_{gen} \circ match_{tycon}(\tau_l, \tau_q)$
Assume $match_{tycon} \circ match_{gen}(\tau_l, \tau_q)$.
Then $\exists\ V_{TC}, U : match_E(V_{TC} \circ U(\tau_l), \tau_q)$, where $U = [\tau_1/v_1] \ldots [\tau_n/v_n]$.
Let $V'_{TC} = V_{TC}$, $U' = [V_{TC}\ \tau_1/v_1] \ldots [V_{TC}\ \tau_n/v_n]$.
We can prove $V_{TC} \circ U(\tau_l) = U' \circ V'_{TC}(\tau_l)$ by induction on the lengths of U and V_{TC}.
So $\exists\ U', V'_{TC}$ such that $match_E(U' \circ V'_{TC}(\tau_l), \tau_q)$.
Thus, $match_{gen} \circ match_{tycon}(\tau_l, \tau_q)$.

(b) $match_{gen} \circ match_{tycon}(\tau_l, \tau_q) \Rightarrow match_{tycon} \circ match_{gen}(\tau_l, \tau_q)$
Assume $match_{gen} \circ match_{tycon}(\tau_l, \tau_q)$.
Then $\exists\ U, V_{TC}$ such that $match_E(U \circ V_{TC}(\tau_l), \tau_q)$,
where $V_{TC} = [X_1/Y_1] \ldots [X_n/Y_n]$ and $U = [\tau_1/v_1] \ldots [\tau_n/v_n]$.
Let V_Y = a sequence of type constructor renamings. For each $[X_i/Y_i] \in V_{TC}$, if $Y_i \in U$ then $[Z_i/Y_i] \in V_Y$, where Z_i is a new type constructor.
V_Y^{-1} = the inverse of V_Y
V_X^{-1} = a sequence of type constructor renamings. For each $[X_i/Y_i] \in V_{TC}$, if $X_i \in U$ then $[Y_i/X_i] \in V_X^{-1}$
$U' = (V_X^{-1} \circ V_Y)(U)$ (i.e., $U' = [V_X^{-1} \circ V_Y \tau_1/v_1] \ldots [V_X^{-1} \circ V_Y \tau_n/v_n]$)
$V'_{TC} = V_Y^{-1} \circ V_{TC}$
With some careful symbol manipulation, we can prove that $U \circ V_{TC}(\tau_l) = U' \circ V'_{TC}(\tau_l)$.
So $\exists\ U', V'_{TC}$ such that $match_E(V'_{TC} \circ U'(\tau_l), \tau_q)$.
Thus, $match_{tycon} \circ match_{gen}(\tau_l, \tau_q)$. □

Informally, (1), (2), and (3) are commutative because the transformations are applied to different types and do not interact. (4) and (5) are commutative because the names of type constructors are independent of the ordering of elements of a tuple or the "curried-ness" of a function. Renaming a type constructor cannot introduce new elements into a tuple and vice versa (and similarly for curried-ness).

Proving commutativity of type constructor renaming and generalized match (6) is more difficult, because the type constructor renamings and the variable substitutions interact, so it is not true that for an arbitrary type constructor renaming, V_{TC}, and variable substitution, U, $V_{TC} \circ U(\tau) = U \circ V_{TC}(\tau)$. For example, if $V_{TC} = [C/T], U = [\beta\ T/\alpha]$, and $\tau = \alpha$, then $V_{TC} \circ U(\tau) = \beta\ C$, while $U \circ V_{TC}(\tau) = \beta\ T$.

However, given a type constructor renaming, V_{TC}, and a variable substitution, U, we can construct a new type constructor renaming, V'_{TC}, and a new substitution, U', that are applied in the opposite order to give the same result (i.e., for (a), $V_{TC} \circ U(\tau) = U' \circ V'_{TC}(\tau)$ and for (b), $U \circ V_{TC}(\tau) = V'_{TC} \circ U'(\tau)$). The construction of V'_{TC} and U' varies in the two cases. In (a), the type constructor renaming stays the same and U' is the same as U except that V_{TC} is applied to each τ_i. In (b), U' is like U except that it "protects" type constructors that should not be renamed by V_{TC} and "unrenames" type constructors that V_{TC} will rename. V'_{TC} applies V_{TC} and then "unprotects" the type constructors protected by U'. For example, let $\tau_l = \beta \rightarrow \gamma\ C$, $\tau_q = \alpha\ T * \alpha\ C \rightarrow \gamma\ T$. Then $match_{gen} \circ match_{tycon}(\tau_l, \tau_q)$ with $U = [\alpha\ T * \alpha\ C/\beta]$ and $V_{TC} = [T/C]$. $U' = [\alpha\ C * \alpha\ Z/\beta]$ (C is renamed to Z to "protect" it from being renamed; T is "unrenamed" to C, since otherwise V_{TC} could not be applied after U). $V'_{TC} = [C/Z][T/C]$ (C is "unprotected" after the renamings from V_{TC}).

The remaining four pairs of matches are not commutative. For each pair, there is one ordering that admits more matches than the other. The following theorem enumerates these relationships.

Theorem 2.4.3 *The following match compositions are not commutative, but the implications hold:*

1. $match_{gen} \circ match_{reorder}(\tau_l, \tau_q) \Rightarrow match_{reorder} \circ match_{gen}(\tau_l, \tau_q)$
2. $match_{gen} \circ match_{uncurry}(\tau_l, \tau_q) \Rightarrow match_{uncurry} \circ match_{gen}(\tau_l, \tau_q)$
3. $match_{spcl} \circ match_{uncurry}(\tau_l, \tau_q) \Rightarrow match_{uncurry} \circ match_{spcl}(\tau_l, \tau_q)$
4. $match_{uncurry} \circ match_{reorder}(\tau_l, \tau_q) \Rightarrow match_{reorder} \circ match_{uncurry}(\tau_l, \tau_q)$

We demonstrate non-commutativity by identifying a τ_l and τ_q such that one match holds but the other does not.

(1) Let $\tau_q = (bool * int) \rightarrow (int * bool)$ and $\tau_l = \alpha \rightarrow \alpha$. Then $(match_{gen} \circ match_{reorder})(\tau_l, \tau_q)$ is false, but $(match_{reorder} \circ match_{gen})(\tau_l, \tau_q)$ is true with the substitution $[(int * bool)/\alpha]$ and a permutation that swaps the order of a 2-element tuple. In the second case, we can apply the reordering *after* we have substituted in type expressions that contain a tuple. However, in the first case, variable substitution comes last, so there is no way to reorder any tuples introduced by the substitution.

(2 and 3) Let $\tau_l = int \rightarrow \alpha$ and $\tau_q = (int * int) \rightarrow int$. Then $match_{uncurry} \circ match_{gen}(\tau_l, \tau_q)$ is true (as is $match_{uncurry} \circ match_{spcl}(\tau_q, \tau_l)$). But $match_{gen} \circ match_{uncurry}(\tau_l, \tau_q)$ is false (as is $match_{spcl} \circ match_{uncurry}(\tau_q, \tau_l)$). In the true case, variable substitution may introduce a functional return value that is then uncurried, while in the false case, it is not possible to instantiate the variable to the same form.

(4) Let $\tau_l = int \rightarrow bool \rightarrow int$ and $\tau_q = bool * int \rightarrow int$. $match_{reorder} \circ match_{uncurry}(\tau_l, \tau_q)$ is true, but $match_{uncurry} \circ match_{reorder}(\tau_l, \tau_q)$ is false. Uncurrying may introduce tuples that are then reordered in the first case but cannot be reordered in the second case.

Proofs of the implications show that the same or slightly modified transformations from the stronger match can be used to show the weaker match as well. For example, consider (1). For any τ_l and τ_q such that $(match_{gen} \circ match_{reorder})(\tau_l, \tau_q)$, we can use the same substitution and tuple transformation to get $(match_{reorder} \circ match_{gen})(\tau_l, \tau_q)$, since variable substitution cannot change the number of elements in a tuple.

Composing a match with itself

Applying the same match twice does not make a difference. This is similar to the notion of idempotence for an operation, but is for relations, and there is the possibility that in the case where we compose the match with itself there are two different transformations.

Theorem 2.4.4 *For $R \in \{tycon, reorder, uncurry, gen, spcl\}$,*
$match_R \circ match_R(\tau_l, \tau_q) \Leftrightarrow match_R(\tau_l, \tau_q)$.

Proof Sketch

$\Leftarrow$: Assume $match_R(\tau_l, \tau_q)$. Then $\exists\, T : match_E(T_l(\tau_l), T_q(\tau_q))$. To show $match_R \circ match_R(\tau_l, \tau_q)$, we need a transformation pair T' such that $match_E(T'_l \circ T_l(\tau_l), T'_q \circ T_q(\tau_q))$. For $R \neq$ uncurry, let $T'_l = T'_q = Id$ (the identity function). For $R =$ uncurry, $T'_l = T'_q = UC$, so we need to show that $UC(UC(\tau)) = UC(\tau)$, which follows by cases from the definition of UC.

$\Rightarrow$: Assume $match_R \circ match_R(\tau_l, \tau_q)$. Then $\exists\, T1,\ T2 : match_E(T1_l \circ T2_l(\tau_l), T1_q \circ T2_q(\tau_q))$. To show $match_R(\tau_l, \tau_q)$, we need a transformation pair T' such that $match_E(T'_l(\tau_l), T'_q(\tau_q))$. For $R =$ uncurry, we again need the fact that $UC(UC(\tau)) = UC(\tau)$ (as shown above). For $R =$ reorder, we use function composition to construct a new permutation $\sigma' = \sigma_1 \circ \sigma_2$. For $R \in$ {gen, spcl, tycon}, we concatenate renamings (e.g., $U' = U1\,\hat{}\,U2$). □

Composed Equivalence and Partial Orders

Composing an equivalence match with another equivalence match yields an equivalence match. We have proved the following composed equivalence matches. Proofs are straightforward manipulations of transformations.

Theorem 2.4.5 *The following are equivalence matches:*

1. $match_{tycon} \circ match_{reorder}$
2. $match_{tycon} \circ match_{uncurry}$
3. $match_{reorder} \circ match_{uncurry}$
4. $match_{tycon} \circ match_{reorder} \circ match_{uncurry}$

Composing an equivalence match with a partial order match yields a partial order match. Again, proofs are by manipulation of transformations.

Theorem 2.4.6 *Let $match_R$ = any subsequence of the sequence $match_{tycon} \circ match_{reorder} \circ match_{uncurry}$. Then $match_R \circ match_{gen}$ is a partial order match and $match_R \circ match_{spcl} \circ match_{uncurry^+}$ is a partial order match.*

Composing generalized and specialized match yields a match that is neither an equivalence match nor a partial order match, because the composed match is not transitive. Consider the following counterexample. Let $\tau_1 = int \rightarrow int$, $\tau_2 = \alpha \rightarrow \alpha$, and $\tau_3 = bool \rightarrow bool$. Then $match_{gen} \circ match_{spcl}(\tau_1, \tau_2)$ and $match_{gen} \circ match_{spcl}(\tau_2, \tau_3)$, but not $match_{gen} \circ match_{spcl}(\tau_1, \tau_3)$.

2.4.3 Relating the Matches

An important property of the relaxed matches and their compositions is monotonicity: adding a new relaxation to a match produces a superset of the existing matches. This property is important for both retrieval and indexing applications. For retrieval, if we make a query with a set of relaxations, we know that (1) we will not lose any matches if we add a relaxation, and (2) we will not add any matches if we remove a relaxation. For indexed libraries, this property means that if an index uses the most relaxed match, we are assured that even using fewer relaxations, the components matching a query must all be in the same node of the index.

We prove this property by considering two cases: first, the relationship between exact match and any of the relaxed matches (Theorem 2.4.7), and second, the relationship between a composed match and the result of adding one more relaxation to that match (Theorem 2.4.8).

Theorem 2.4.7 *For $R \in \{tycon, reorder, uncurry, gen, spcl\}$,*

$$match_E(\tau_l, \tau_q) \Rightarrow \mathrm{match}_R(\tau_l, \tau_q)$$

Proof For $R \neq$ uncurry, let $T_l = T_q = Id$ (the identity function). $match_E(\tau_l, \tau_q) \Rightarrow match_E(T_l\tau_l, T_q\tau_q)$, so $match_R(\tau_l, \tau_q)$.

For $R =$ uncurry, we can show that $\tau_l =_T \tau_q \Rightarrow UC(\tau_l) =_T UC(\tau_q)$ from the definition of UC. □

Theorem 2.4.8 *Let M be a subsequence of $(match_{tycon} \circ match_{reorder^+} \circ match_{uncurry^+} \circ match_{gen} \circ match_{spcl})$, where R is the set of relaxations used in M. Let $M1$ be the result of adding one more relaxation (in this order) to M, where $R1$ is the set of relaxations used in $M1$ (so $|R1| = |R|+1$). Then*

$$M(\tau_l, \tau_q) \Rightarrow M1(\tau_l, \tau_q)$$

Proof Sketch Let $X = R1 - R$ (i.e., X is the added relaxation). Assume $M(\tau_l, \tau_q)$.

Case 1: $X \neq$ uncurry, let $T_l = T_q = Id$ (the identity function). Then $T_l\tau_l = \tau_l$ and $T_q\tau_q = \tau_q$, so $M(\tau_l, \tau_q) \Rightarrow M1(T_l\tau_l, T_q\tau_q)$, and (using the same transformations) $M1(\tau_l, \tau_q)$.

Case 2: $X =$ uncurry.

Case 2a: tycon, reorder $\notin R$. Then uncurry is applied after any other transformations.

$M(\tau_l, \tau_q) \Rightarrow \exists\ T_l, T_q : match_E(T_l\tau_l, T_q\tau_q) \Rightarrow match_E(UC(T_l\tau_l), UC(T_q\tau_q)) \Rightarrow match_{uncurry} \circ M(\tau_l, \tau_q) \Rightarrow M1(\tau_l, \tau_q)$.

Case 2b: tycon $\in R$, reorder $\notin R$. Uncurrying does not change type constructors, so we can uncurry before renaming type constructors and still match. (Full proof by cases using definitions of UC and type constructor renaming.)

Case 2c: reorder $\in R$, tycon $\notin R$. $M(\tau_l, \tau_q) \Rightarrow \exists\ T_\sigma, T_l, T_q : match_E(T_\sigma \circ T_l\tau_l, T_q\tau_q)$.
Make $T_{\sigma'}$ an extension of T_σ to account for uncurrying, so that $match_E(T_{\sigma'}(UC(T_l\tau_l)), UC(T_q\tau_q))$.
Case 2d: reorder, tycon $\in R$. The analysis is similar to that for cases 2b and 2c. $\Box$

2.4.4 Generic Match Forms

Recall the generic match definition (Definition 2.2.1):

$$M(\tau_l, \tau_q) = \exists \text{ a } \textit{transformation pair},\ T = (T_l, T_q), \text{ such that } T_l(\tau_l)\ \mathcal{R}\ \ T_q(\tau_q)$$

Table 2.2 shows how $\mathcal{R}$ is instantiated and the kinds of transformations in T_l and T_q for each of the basic function match definitions presented in this section as well as unification, and two of the combined matches to show how the matches can be combined. The relation $\mathcal{R}$ is either $=_T$, $\geq$, $\leq$, or exact function match ($match_E$). The transformations T_l and T_q are one of Id (the identity function), V (variable renaming), U (substitution), V_{TC} (type constructor renaming), T_σ (permute tuple) or UC (uncurry).

Match	$\mathcal{R}$	T_l	T_q
Exact	$=_T$	V	Id
Type Constructor	$match_E$	V_{TC}	Id
Uncurry	$match_E$	UC	UC
Reorder	$match_E$	T_σ	Id
Generalized	$\geq$	Id	Id
Generalized (alternative)	$=_T$	U	Id
Specialized	$\leq$	Id	Id
Specialized (alternative)	$=_T$	Id	U
Unification	$=_T$	U	U
Reorder $\circ$ Uncurry	$match_E$	$T_\sigma \circ UC$	UC
Uncurry $\circ$ Generalized	$match_E$	$UC \circ\ U$	UC

Table 2.2: Instantiations of generic function match

2.5 Implementation

We have implemented a function signature matcher for Standard ML (SML) functions and incorporated it into both a signature-based retrieval tool and an index builder. Everything is implemented in SML as well. The SML type system has some features that we did not discuss in the definitions because they are slightly unusual and not central to the notion of type matching. In the implementation, we handle these features in as simple a way as possible: two record types match if the field names are the same and the types of each field match; we do not distinguish between eq types and non-eq types (eq types are polymorphic types that can be compared for equality); and *ref* is treated as a regular type constructor (a value with type τ *ref* is a storage location for a value of type τ; the stored value can be modified).

We designed the signature matcher to allow us to experiment easily with different relaxed matches and combinations of relaxed matches. The matcher has a parameter that specifies which relaxations to use and thus the user can "pick and choose" from among five relaxations (generalized, specialized, reorder, uncurry, and type constructor). The order in which relaxations are composed in the implementation is the same as that discussed in the examples of Section 2.3. When all relaxations are selected, the match is:

$$(match_{tycon} \circ \ match_{reorder^+} \circ \ match_{uncurry^+} \circ \ match_{gen} \circ \ match_{spcl})(\tau_l, \tau_q)$$

For any subset of relaxations selected, the relative ordering remains the same. For example, if only the reorder and generalized relaxations are selected, the match is $(match_{reorder^+} \circ match_{gen})(\tau_l, \tau_q)$. We say equivalently that this is "match with relaxations reorder and generalized". Thus, the matcher implements exact match, each of the relaxed matches, and the composed matches.

The implementation uses the recursive versions of reorder and uncurry ($match_{reorder^+}$ and $match_{uncurry^+}$). The algorithms for the generalized, specialized, and type constructor relaxations are modifications of Robinson's unification algorithm, as presented by Milner [Mil78]. The algorithms for the other relaxations are straightforward; we use a simple transformation of the type for the uncurry relaxations, and a backtracking algorithm for the reorder relaxation. We can use a single match implementation and pick and choose relaxations because each relaxation affects different aspects of the type. Tuple reordering affects only tuples, uncurrying affects only higher-order function application, and type constructor renaming affects only type constructors. Both generalized and specialized match affect type variables, but their interaction is limited, since generalized match instantiates variables in τ_l and specialized match instantiates variables in τ_q.

Without any relaxations, the signature matching algorithm is linear in the size of the type. Allowing both variable instantiation and tuple reordering means that we must allow backtracking in the algorithm. For example, $(int * bool)$ matches with $(\alpha * int)$ under the relaxations generalized and reorder by instantiating α to *bool*. But the first attempt to match the types would instantiate α to *int* and then fail, so we must be able to backtrack. With reordering and generalized or specialized relaxations, matching thus becomes exponential in the size of the tuples in the types. In practice, most tuples have only two or three elements, so the match is still efficient. We do not expect the generalized and specialized relaxations to be used together, so we have not analyzed the complexity of a match with both these relaxations.

2.5.1 Beagle: Signature-based Retrieval

We used the function signature matcher to implement a signature-based retrieval tool called *Beagle*.[2] Given a query and a set of relaxations, Beagle uses the appropriate match to compare each function in the library with the query and returns the set of functions from the library that match with the query. Beagle can use any library of SML functions, but in our examples, we use the test library described in Section 1.2.3, which contains 1451 SML functions. Chapter 5 (in particular, Section 5.1) gives many example uses of Beagle. On a test suite of 10 queries on each of the 16 combinations of the generalized, specialized, reorder, and uncurry relaxations using our library, the average time to complete a query was .13 seconds, ranging from averages of .08 seconds for exact match to .19 seconds for the match using all 4 relaxations.

Beagle's user interface is intentionally simplistic – it is just gnu-emacs [Sta86] and a mouse. The user defines the query and selects the desired relaxations before performing a search. The output is a list of functions whose types match the query, along with the pathname for the file that contains the function. Figure 2.1 shows the results of a query as they appear in the emacs buffer (`'a` and `'b` denote type variables). Clicking the mouse on a function in the list causes the file in which the function is defined to appear in another buffer, with the cursor located at the beginning of the function definition. We chose to use emacs for our interface rather than a flashier graphical user interface in order to give programmers easy access to signature-based retrieval from their normal software development environment. We wanted to make signature-based retrieval as easily available for use as string searching.

2.5.2 Index Builder

We also used the function signature matching package to implement an *index builder*, a tool to build an indexed library from a library of components. Section 5.2 describes indexed libraries in detail. The index builder takes as input a component library and a pair of matches, where the matches define equivalence and partial ordering on components. The output of the index builder is a directed acyclic graph in which each node contains an equivalence class of functions, and edges are directed by the partial ordering relation. The implementation is completely parameterized by the kind of elements in the library and the pair of matches, but examples in Section 5.2 are the result of instantiating the library and match pair by a function signature library and function signature matches.

[2]Beagles are hunting dogs that are well-known for their ability to find animals by following a scent.

```
Query = (('a list * 'b list) -> ('a * 'b) list)
Matcher = Curry
Total number of objects found: 3
---------------------------------------------------------------
zip :(('a list * 'b list) -> ('a * 'b) list)
^    37 /usr/misc/.sml/lib/edinburgh/portable/ListPair.sml

zip :(('a list * 'b list) -> ('a * 'b) list)
^    54 /usr/misc/.sml/lib/smlnj-lib/list-util.sml

zip :('a list -> ('b list -> ('a * 'b) list))
^    10 /usr/misc/.sml/local/lib/Container/listFns.sml

---------------------------------------------------------------
```

Figure 2.1: Output buffer of Beagle

2.5.3 Why use ML?

Implementing signature matching and Beagle for ML gave us the opportunity to explore interesting relaxed matches and also the composition of relaxed matches. A pleasant by-product of the decision to use ML is that it led naturally to using ML to implement signature matching, the retrieval tool (Beagle), and the index builder. We believe that using ML led to an easier and cleaner implementation. For example, it took only a few days to implement the index builder, which consists of two modules (one for the node type and operations, and the other to build a directed acyclic graph of nodes).

There were two drawbacks to our choice of ML. First, the type system is, in a sense, too rich. It includes features like equality types and weak type variables, which are not common features in type systems. We chose to exclude these features in order to have more general function match definitions. A second drawback was the relatively small user community. While the use of ML is growing, it is not yet a rival to C or C++ in number of users. This arose as a problem when we were generating a component library and trying to evaluate the usefulness of signature-based retrieval. A small user community means having fewer sources of components that could be used as a library. The Community Library is large enough to preclude easy retrieval of components by inspection, but since it is composed of three smaller, somewhat overlapping libraries that are fairly well-known to local ML users, there was little incentive for them to use Beagle. A larger more diverse library would have made Beagle a more useful resource.

2.6 Discussion

In this section, we discuss what the advantages of signature matching are, how the approach applies to less-expressive type systems, why we chose to define the matches in terms of transformations and to separate out each relaxation, and what some alternative matches and approaches are. Much of our evidence of the usefulness of function signature matching and of which combinations of relaxations are most likely to be used in practice is based on our experience in using signature matching for retrieval and indexing. Thus, we refer the reader to our discussion of applications in Chapter 5 for examples of the use of signature matching and for recommendations about which relaxations to use for each application.

Regardless of the application, signatures have a number of features that make them a good choice as a method of comparing software components. First, signatures already exist for library components, since they are either generated automatically by type inference systems, or they are provided programmers for the compiler anyway. Second, using signatures for retrieval means that users write queries in a language they already understand. Third, implementing signature matching requires nothing more sophisticated than unification, a standard algorithm already used in some compilers to do type inference. Thus, signatures offer useful semantic information and are relatively "cheap" in terms of overhead.

The richness of a language's type system affects how well signature matching works. However, even for languages with a fairly basic type system a function's type carries a lot of information about the function, so the approach still applies, as do the main match definitions. Obviously, if a language does not allow higher-order functions, then the uncurry relaxation does not make sense. Similarly, for a language without user-defined types the type constructor relaxation does not make sense. However, even for a language with a non-polymorphic type system, we can still use specialized match by extending the query language to include type variables.

Our approach to signature matching is to define each match in terms of a transformation. Each of our match transformations produces a "wrapper" that transforms the type in some way. For reorder and uncurry match, wrappers rearrange the form of a type (reordering tuple elements or (un)currying). For the other matches, wrappers rename or instantiate type variables or type constructors. When we use signature matching to retrieve a library function or to compare functions in order to substitute one for the other, a function is more useful if the overhead of using it is relatively low, i.e., if the component can be used directly, perhaps even automatically. Having wrappers makes this possible.

Using transformations lets us define the composition of matches in terms of composition of the transformation functions, and to prove the various theorems about properties of composed matches. The different kinds of transformations (e.g., variable substitutions, permutations) and the possibility of interaction between transformations makes it necessary to be very careful in doing the proofs.

Using transformations also lets us define each relaxation separately in terms of a transformation. We find separating the relaxations to be a very important factor in our approach. Which match definition is appropriate varies depending on the application. For example, the match needed to retrieve a function in the library to concatenate strings is not the same as the match needed to find out how many functions in the library have a two-element tuple as their input. In the first case, we do not care about curriedness, but we know we cannot use a more general function, so we use uncurry match with the query $string * string \rightarrow string$; in the second case, we do not care about the actual types, only the form, so we use specialized match with the query $\alpha * \beta \rightarrow \gamma$.

Using transformations and defining each relaxation separately within a generic form also allows us to consider new relaxations easily. There were some other relaxed matches that we considered but do not include because they did not generate wrappers that would allow one function to be used directly in place of the matching function. One such relaxation is to allow matches where two tuples have a different number of arguments. For example, a function *get-n* that returns n elements of a list starting with the m^{th} element, of type $int * int * \alpha\ list \rightarrow \alpha\ list$, would match with a function *first-n* that returns the first n elements of a list, of type $int * \alpha\ list \rightarrow \alpha\ list$. We can easily define this in a manner similar to reorder match, where the permutation is not required to be a bijection. However, a wrapper that removes an element from a tuple or adds an element to a tuple would also need to know something more about the relationship between the two types in order to work properly (e.g., to implement *first-n* using *get-n*, the wrapper would need to instantiate the first argument to *get-n* with 0). This knowledge cannot be determined from just the signatures. Note that a user of a retrieval system could still find matches like this with a small sequence of queries, which allows the user far greater control in determining which arguments are essential and which can be dropped. Another relaxed match we did not include was one that "flattens" nested tuples. With this match, the type $((int * bool) * int)$ would match $(int * bool * int)$. We do not include this match because in actual code, tuples are usually nested for a reason, and thus, it would be unusual to want to treat them as being the same.

The examples of relaxed matches we did not include illustrates a disadvantage to our approach: there will always be more relaxations that we could consider. An alternative approach to defining function signature match uses category theory to define isomorphisms of types [Rit90, DC92]. The advantage to this approach is that the theory is complete. However, the approach is not as flexible as using transformations, since it does not allow each relaxation to be selected separately – there is a single match definition. Additionally, some of the axioms required for completeness give rise to unintuitive isomorphisms (e.g., $unit * \alpha$ is isomorphic to α).

Chapter 3

Function Specification Matching

In this chapter, we define function specification matching. We begin by briefly describing Larch/ML, the specification language we use, in Section 3.1. Section 3.2 presents the various exact and relaxed specification match definitions. We summarize properties of the matches and how they relate to each other in Section 3.3, and describe our implementation of a function specification matcher in Section 3.4.

3.1 Larch/ML Specifications

We use Larch/ML [WRZ93], a Larch interface language for the ML programming language, to specify ML functions and ML modules. Larch provides a "two-tiered" approach to specification [GH93]. In one tier, the specifier writes *traits* in the Larch Shared Language (LSL) to assert state-independent properties. Each trait introduces *sorts* and *operators* and defines equality between terms composed of the operators (and variables of the appropriate sorts). Appendix A shows the *Container* trait, which defines operators to generate containers (*empty* and *insert*), to return the element or container resulting from deleting an element from the beginning or end (*first*, *last*, *butFirst*, and *butLast*), to return the length of a container (*length*), and to determine whether a container is empty (*isEmpty*).

In the second tier, the specifier writes *interfaces* in a Larch interface language to describe state-dependent effects of a program (see Figure 3.1). The Larch/ML interface language extends ML by adding specification information in special comments delimited by (*+...+*). The **using** and **based on** clauses link interfaces to LSL traits by specifying a correspondence between (programming-language specific) types and LSL sorts. For polymorphic sorts, there must be an associated sort for both the polymorphic variable (e.g., α) and the type constructor (e.g., T) in the **based on** clause. The specification for each function begins with a *call pattern* consisting of the function name followed by a pattern for each parameter, optionally followed by an equal sign (=) and a pattern for the result. In ML, patterns are used in binding constructs to associate names to parts of values. For example, *(x, y)* names x as the first of a pair and

signature Stack = **sig**
  (*+ **using** Container +*)
  type $\alpha\ T$ (*+ **based on**
    Container.E Container.C +*)

  val $create$: $unit \rightarrow \alpha\ T$
  (*+ $create$ ( ) = s
    ensures $s = empty$ +*)

  val $push$: $\alpha\ T * \alpha \rightarrow \alpha\ T$
  (*+ $push$ $(s, e) = s2$
    ensures $s2 = insert(e, s)$ +*)

  val pop : $\alpha\ T \rightarrow \alpha\ T$
  (*+ pop $s = s2$
    requires $not(isEmpty(s))$
    ensures $s2 = butLast(s)$ +*)

  val top : $\alpha\ T \rightarrow \alpha$
  (*+ top $s = e$
    requires $not(isEmpty(s))$
    ensures $e = last(s)$ +*)
end

signature Queue = **sig**
  (*+ **using** Container +*)
  type $\alpha\ T$ (*+ **based on**
    Container.E Container.C +*)

  val $create$: $unit \rightarrow \alpha\ T$
  (*+ $create$ ( ) = q
    ensures $q = empty$ +*)

  val enq : $\alpha\ T * \alpha \rightarrow \alpha\ T$
  (*+ enq $(q, e) = q2$
    ensures $q2 = insert(e, q)$ +*)

  val $rest$: $\alpha\ T \rightarrow \alpha\ T$
  (*+ $rest$ $q = q2$
    requires $not(isEmpty(q))$
    ensures $q2 = butFirst(q)$ +*)

  val deq : $\alpha\ T \rightarrow \alpha$
  (*+ deq $q = e$
    requires $not(isEmpty(q))$
    ensures $e = first(q)$ +*)
end

Figure 3.1: The Toy Specification Library (Larch/ML modules)

y as the second. The **requires** clause specifies the function's pre-condition as a predicate in terms of trait operators and names introduced by the call pattern. Similarly, the **ensures** clause specifies the function's post-condition. If a function does not have an explicit **requires** clause, the default pre-condition is *true*. A function specification may also include a **modifies** clause, which lists those objects whose values may change as a result of executing the function. Larch/ML also includes rudimentary support for specifying higher-order functions.

The Larch/ML interface specifications in Figure 3.1 are the Toy Specification Library, which we use in our examples of specification matching. The library contains two module specifications: one for *Stack* with the functions *create*, *push*, *pop*, and *top*, and one for *Queue*, with the functions *create*, *enq*, *rest*, and *deq*. We specify each function's pre- and post-conditions in terms of operators from the *Container* trait.

3.2 Match Definitions

For a function specification, S, we denote the pre- and post-conditions as S_{pre} and S_{post}, respectively. S_{pred} defines the interpretation of the function's specification as an implication between the two: $S_{pred} = S_{pre} \Rightarrow S_{post}$. Intuitively, this interpretation means that if S_{pre} holds when the function specified by S is called, S_{post} will hold after the function has executed (assuming the function terminates). If S_{pre} does not hold, there are no guarantees about the behavior of the function. This interpretation of a pre- and post-condition specification is the most common and natural for functions in the standard programming model. For example, for the Stack *top* function in Figure 3.1

- The pre-condition, top_{pre}, is $not(isEmpty(s))$.
- The post-condition, top_{post}, is $e = last(s)$.
- The specification predicate, top_{pred}, is $(not(isEmpty(s))) \Rightarrow (e = last(s))$.

To be consistent in terminology with Chapter 2, we present function specification matching in the context of a retrieval application. Example matches are between a specification S from the Toy Specification Library in Figure 3.1 and a query specification Q. We assume that variables in S and Q have been renamed consistently. This renaming is easily provided by the signature matcher, and we are assuming that the signatures of S and Q match. For example, if we compare the Stack *pop* function with the Queue *rest* function, we must rename q to s and $q2$ to $s2$. In this section we examine several definitions of the specification match predicate. We characterize definitions as either grouping pre-conditions S_{pre} and Q_{pre} together and post-conditions S_{post} and Q_{post} together, or relating predicates S_{pred} and Q_{pred}. Both of these kinds of matches have a general form.

Definition 3.2.1 (Generic Pre/Post Match)

$$match_{pre/post}(S, Q) = (Q_{pre}\ \mathcal{R}_1\ S_{pre})\ \mathcal{R}_2\ (S_{post}\ \mathcal{R}_3\ Q_{post})$$

Pre/post matches relate the pre-conditions of each component and the post-conditions of each component. Post-conditions of related functions are often similar. For example, they may specify related properties of the return values. Similarly, pre-conditions of related functions may specify related bounds conditions of input values. The relations $\mathcal{R}_1$ and $\mathcal{R}_3$ relate pre-conditions and post-conditions respectively, and hence are either equivalence ($\Leftrightarrow$) or implication ($\Rightarrow$), but need not be the same. In most cases we require that both relations ($\mathcal{R}_1$ and $\mathcal{R}_3$) hold, and so $\mathcal{R}_2$ is usually conjunction ($\wedge$) but may also be implication ($\Rightarrow$). The matches may vary from this form by dropping some of the terms. Table 3.1 summarizes how $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ are instantiated for each of the matches in Section 3.2.1. For example, $match_{plug\text{-}in} = (Q_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow Q_{post})$. For $match_{plug\text{-}in\text{-}post}$ and $match_{weak\text{-}post}$, $\mathcal{R}_1$ is not instantiated

Match	$\mathcal{R}_1$	$\mathcal{R}_2$	$\mathcal{R}_3$
Exact Pre/Post	$\Leftrightarrow$	$\wedge$	$\Leftrightarrow$
Plug-in	$\Rightarrow$	$\wedge$	$\Rightarrow$
Plug-in Post	drop S_{pre} and Q_{pre}	—	$\Rightarrow$
Weak Post	drop Q_{pre}	$\Rightarrow$	$\Rightarrow$

Table 3.1: Instantiations of generic pre/post match $((Q_{pre}\ \mathcal{R}_1\ S_{pre})\ \mathcal{R}_2\ (S_{post}\ \mathcal{R}_3\ Q_{post}))$

because one or both of its arguments are dropped. Similarly, for $match_{plug\text{-}in\text{-}post}$, $\mathcal{R}_2$ is not instantiated because both Q_{pre} and S_{pre} are dropped.

Definition 3.2.2 (Generic Predicate Match)

$$match_{pred}(S, Q) = S_{pred}\ \mathcal{R}\ Q_{pred}$$

Predicate matches relate the entire specification predicates, S_{pred} and Q_{pred}, of the two components. The relation $\mathcal{R}$ is either equivalence ($\Leftrightarrow$), implication ($\Rightarrow$), or reverse implication ($\Leftarrow$). Table 3.2 summarizes how $\mathcal{R}$ is instantiated for each of the matches in Section 3.2.2. Predicate matches are useful in cases where we need to consider the relationship of the specifications as a whole rather than relationships of the parts, for example, when we need to assume something from the pre-condition in order to reason about post-conditions.

Match	$\mathcal{R}$
Exact Predicate	$\Leftrightarrow$
Generalized	$\Rightarrow$
Specialized	$\Leftarrow$

Table 3.2: Instantiations of generic predicate match $(S_{pred}\ \mathcal{R}\ Q_{pred})$

It is important to look at both kinds of match. Which kind of match is appropriate may depend on the context in which the match is being used or on the specifications being compared. We present the pre/post matches in Section 3.2.1 and the predicate matches in Section 3.2.2. For each, we present a notion of exact match as well as relaxed matches.

3.2.1 Pre/Post Matches

Pre/post matches on specifications S and Q relate S_{pre} to Q_{pre} and S_{post} to Q_{post}. We consider four kinds of pre/post matches, beginning with the strongest match and progressively weakening the match by relaxing the relations $\mathcal{R}_1$ and $\mathcal{R}_3$ from $\Leftrightarrow$ to $\Rightarrow$, by relaxing $\mathcal{R}_2$ from $\wedge$ to $\Rightarrow$, or by dropping one or more terms.

Exact Pre/Post Match

We begin by instantiating both $\mathcal{R}_1$ and $\mathcal{R}_3$ to $\Leftrightarrow$ and $\mathcal{R}_2$ to $\wedge$ in the generic pre/post match of Definition 3.2.1. Two function specifications satisfy the *exact pre/post match* if their pre-conditions are equivalent and their post-conditions are equivalent. If exact pre/post match holds for two specifications, they are essentially equivalent and thus completely interchangeable. Anywhere that one is used, it could be replaced by the other with no change in observable behavior.

Definition 3.2.3 (Exact Pre/Post Match)

$$match_{E\text{-}pre/post}(S,Q) = (Q_{pre} \Leftrightarrow S_{pre}) \wedge (S_{post} \Leftrightarrow Q_{post})$$

Exact pre/post match is a strict relation, yet two different-looking specifications can still satisfy the match. Consider for example the following query $Q1$, based on the *Container* trait. $Q1$ specifies a function that returns a container whose size is zero, one way of specifying a function to create a new container.

$\boxed{\text{Q1}}$

signature Q1 = **sig**
  (∗+ **using** Container +∗)
  type $\alpha\ T$ (∗+ **based on** Container.E Container.C +∗)
  val $qCreate : unit \rightarrow \alpha\ T$
  (∗+ $qCreate$ ( ) = c
   ensures $length(c) = 0$ +∗)
end

Under exact pre/post match, $Q1$ is matched by both the Stack and Queue *create* functions in the Toy Specification Library. (The specifications of Stack and Queue *create* are identical except for the name of the return value.)

Let us look in more detail at how the Stack *create* specification matches with $Q1$. Let S be the specification for Stack *create* and $Q1$ be the query specification with c renamed to s. $S_{pre} = true$, $S_{post} = (s = empty)$. $Q1_{pre} = true$, $Q1_{post} = (length(s) = 0)$. Since both S_{pre} and $Q1_{pre}$ are $true$, showing $match_{E-pre/post}(S, Q1)$ reduces to proving $S_{post} \Leftrightarrow Q1_{post}$, or $(s = empty) \Leftrightarrow (length(s) = 0)$. The "if" case $((s = empty) \Rightarrow (length(s) = 0))$ follows immediately from the axioms in the *Container* trait about *length*. Proving the "only-if" case

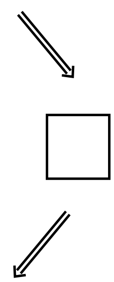


Figure 3.2: Idea behind plug-in match

$((length(s) = 0) \Rightarrow (s = empty))$ requires only basic knowledge about integers and the fact that for any container, s, $length(s) \geq 0$, which is provable from the *Container* trait.

Plug-in Match

Equivalence is a strong requirement. For *plug-in match*, we relax both $\mathcal{R}_1$ and $\mathcal{R}_3$ to $\Rightarrow$ and keep $\mathcal{R}_2$ as $\wedge$ in the generic pre/post match of Definition 3.2.1. Under plug-in match, Q is matched by any specification S whose pre-condition is weaker (to allow at least all the conditions that Q allows) and whose post-condition is stronger (to provide a guarantee at least as strong as Q provides).

Definition 3.2.4 (Plug-in Match)

$$match_{plug\text{-}in}(S, Q) = (Q_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow Q_{post})$$

Plug-in match captures the notion of being able to "plug-in" S for Q, as illustrated in Figure 3.2. A specifier writes a query Q saying essentially:

> *I need a function such that if Q_{pre} holds before the function executes, then Q_{post} holds after it executes (assuming the function terminates).*

With plug-in match, if Q_{pre} holds (the assumption made by the specifier) then S_{pre} holds (because of the first conjunct of plug-in match). Since we interpret S to guarantee that $S_{pre} \Rightarrow S_{post}$, we can assume that S_{post} will hold after executing the plugged-in S. Finally, since $S_{post} \Rightarrow Q_{post}$ from the second conjunct of plug-in match, Q_{post} must hold, as the specifier desired. We say that S is behaviorally equivalent to Q, since we can plug-in S for Q and have the same observable behavior, but this is not a true equivalence because it is not symmetric: we cannot necessarily plug-in Q for S and get the same guarantees.

Consider the following query. $Q2$ is fairly weak specification of an *add* function. It requires that an input container has less than 50 elements, and guarantees that the resulting container is one element longer than the input container.

signature Q2 = **sig** $\boxed{\text{Q2}}$
  (*+ **using** Container +*)
  type $\alpha\ T$ (*+ **based on** Container.E Container.C +*)
  val $add : \alpha\ T * \alpha \rightarrow \alpha\ T$
  (*+ $add\ (q1, e) = q2$
   requires $length(q1) < 50$
   ensures $length(q2) = (length(q1)\ +1)$ +*)
 end

Under exact pre/post match, $Q2$ is not matched by any function in the Toy Specification Library, but under plug-in match, $Q2$ is matched by both the Stack *push* and the Queue *enq* functions. Since *push* and *enq* are identical except for their names and the names of the variables, the proof of the match is the same for both.

The pre-condition requirement, $Q_{pre} \Rightarrow S_{pre}$, holds, since $S_{pre} = true$. To show that $S_{post} \Rightarrow Q_{post}$, we assume S_{post} $(q2 = insert(e, q))$, and try to show Q_{post} $(length(q2) = length(q) + 1)$. Substituting for $q2$ in Q_{post}, we have $length(insert(e, q)) = length(q) + 1$, which follows immediately from the equations for *length*.

Plug-in Post Match

Often we are concerned with only the effects of functions, thus a useful relaxation of the plug-in match is to consider only the post-condition part of the conjunction. Most pre-conditions could be satisfied by adding an additional check before calling the function. *Plug-in post match* is also an instance of the generic pre/post match of Definition 3.2.1, with $\mathcal{R}_3$ instantiated to $\Rightarrow$ but dropping Q_{pre} and S_{pre}.

Definition 3.2.5 (Plug-in Post Match)

$$match_{plug\text{-}in\text{-}post}(S, Q) = (S_{post} \Rightarrow Q_{post})$$

Consider the following query. $Q3$ is identical to Stack *top* except that $Q3$ has no **requires** clause.

$\boxed{\text{Q3}}$

signature Q3 = **sig**
  (*+ **using** Container +*)
  type $\alpha\ T$ (*+ **based on** Container.E Container.C +*)
  val $delete : \alpha\ T \rightarrow \alpha$
  (*+ $delete\ s = e$
    ensures $e = last(s)$ +*)
end

Stack *top* does not match with $Q3$ under either exact pre/post or plug-in match, since $Q3$'s pre-condition is weaker than Stack *top*'s. Since the post-conditions are equivalent, Stack *top* does match with $Q3$ under plug-in post match.

Weak Post Match

Finally, consider this even weaker match, *weak post match*. We instantiate $\mathcal{R}_3$ to $\Rightarrow$, as with the plug-in matches, but relax $\mathcal{R}_2$ to $\Rightarrow$ and drop Q_{pre}.

Definition 3.2.6 (Weak Post Match)

$$match_{weak\text{-}post}(S, Q) = S_{pre} \Rightarrow (S_{post} \Rightarrow Q_{post})$$

A more intuitive, equivalent, predicate is $(S_{pre} \wedge S_{post}) \Rightarrow Q_{post}$. Sometimes assuming the pre-condition of S helps in proving the relationship between S_{post} and Q_{post}. We use S_{pre} and not Q_{pre} since S_{pre} is likely to be necessary to limit the conditions under which we try to prove $S_{post} \Rightarrow Q_{post}$. The additional assumption also means that we will have to provide an additional "wrapper" in our code to guarantee S_{pre} before we call the function specified by S.

For example, suppose we wish to find a function to delete from a container using the following query $Q4$:

$\boxed{\text{Q4}}$

signature Q4 = **sig**
  (*+ **using** Container +*)
  type $\alpha\ T$ (*+ **based on** Container.E Container.C +*)
  val $remainder : \alpha\ T \rightarrow \alpha\ T$
  (*+ $remainder\ s = s2$
    ensures $length(s2) = (length(s) - 1)$ +*)
end

$Q4$ describes a function that returns a container whose size is one less than the size of the input container. This is a fairly weak way of describing deletion, since it does not specify which element is removed. But it still gives us a big gain in precision over signature matching. $Q4$

Assume $not(isEmpty(s))$	Assume S_{pre}	(1)
Assume $s2 = butLast(s)$	Assume S_{post}	(2)
$length(s2) = length(s) - 1$	Attempt to prove Q_{post}	(3)
$length(butLast(s)) = length(s) - 1$	Apply (2) to (3)	(4)
Let $s = insert(ec, sc)$	Since s is not empty (1), and s generated by empty and insert	(5)
$length(butLast(insert(ec, sc))) = length(insert(ec, sc)) - 1$	Substitute (5) for s in (4)	(6)
$length(sc) = length(insert(ec, sc)) - 1$	Axioms for $butLast$	(7)
$length(sc) = (length(sc) + 1) - 1$	Axioms for $length$	(8)
$length(sc) = length(sc)$	Axioms for $+$, $-$	(9)

Figure 3.3: Proof sketch of $match_{weak\text{-}post}(pop, Q4)$

would not retrieve other functions with the signature $\alpha\ t \rightarrow \alpha\ t$, for example, a function that reverses or sorts the elements in the container, or removes duplicates.[1]

While intuitively $Q4$ would seem related to Stack *pop* and Queue *rest*, neither *pop* nor *rest* match with $Q4$ under either plug-in or plug-in post match. Consider Stack *pop* (the reasoning is similar for Queue *rest*). We cannot prove $S_{post} \Rightarrow Q_{post}$ (i.e., $(s2 = butLast(s)) \Rightarrow (length(s2) = length(s) - 1)$) for the case where $s = empty$. However, by adding the assumption S_{pre} $(not(isEmpty(s)))$, we are able to show that Stack *pop* matches with $Q4$ under weak post match, as we see in the proof sketch in Figure 3.3.

3.2.2 Predicate Matches

Recall the generic predicate match (Definition 3.2.2):

$$match_{pred}(S, Q) = S_{pred}\ \mathcal{R}\ Q_{pred}$$

where the relation $\mathcal{R}$ is equivalence ($\Leftrightarrow$), implication ($\Rightarrow$), or reverse implication ($\Leftarrow$).

Note that this general form allows alternative definitions of the specification predicates. One alternative is $S_{pred} = S_{pre} \wedge S_{post}$, which is stronger than $S_{pred} = S_{pre} \Rightarrow S_{post}$. This interpretation is reasonable in the context of state machines, where the pre-condition serves as a guard so that a state transition occurs only if the pre-condition holds.

As we did with the generic pre/post match, we consider instantiations of the generic predicate match including an exact match and various relaxations.

Exact Predicate Match

We begin with *exact predicate match.* Two function specifications match exactly if their predicates are logically equivalent (i.e., $\mathcal{R}$ is instantiated to $\Leftrightarrow$). This is less strict than exact

[1]We used Beagle, the signature-based retrieval tool, to find these examples. See Browsing 4 in Section 5.1.3 for details.

pre/post match (Definition 3.2.3), since there can be some interaction between the pre- and post-conditions (i.e., $match_{E\text{-}pre/post} \not\Rightarrow match_{E\text{-}pred}$). In fact, in cases where $S_{pre} = Q_{pre} = true$, exact pre/post and exact predicate matches are equivalent.

Definition 3.2.7 (Exact Predicate Match)

$$match_{E\text{-}pred}(S, Q) = S_{pred} \Leftrightarrow Q_{pred}$$

Our example $Q1$ is still matched by Stack and Queue *create* under exact predicate match, since

$$\begin{aligned} S_{pred} \Leftrightarrow Q_{pred} &= (true \Rightarrow (s = empty)) \Leftrightarrow (true \Rightarrow (length(s) = 0)) \\ &= (s = empty) \Leftrightarrow (length(s) = 0) \end{aligned}$$

which is exactly what we proved to show that $Q1$ is matched by Stack and Queue *create* under exact pre/post match.

Generalized Match

For generalized match, we relax $\mathcal{R}$ in the generic predicate match to $\Rightarrow$. Generalized match works well in the context of queries and libraries: specifications of library functions will be detailed, describing the behavior of the functions completely, but we would like to be able to write simple queries that focus only on the aspect of the behavior that we are most interested in or that we think is most likely to differentiate among functions in the library. Generalized match allows the library specification to be stronger (more general) than the query. Note that generalized match is a weaker match than plug-in match. Also, if we drop the pre-conditions in generalized match, we get plug-in post match.

Definition 3.2.8 (Generalized Match)

$$match_{gen\text{-}pred}(S, Q) = S_{pred} \Rightarrow Q_{pred}$$

For example, consider the following query, which is the same as $Q4$ but with a **requires** clause.

signature Q5 = **sig** $\boxed{\text{Q5}}$
  (*+ **using** Container +*)
  type α T (*+ **based on** Container.E Container.C +*)
  val *remainder* : $\alpha\ T \rightarrow \alpha\ T$
  (*+ *remainder* $s = s2$
    requires $not(isEmpty(s))$
    ensures $length(s2) = (length(s) - 1)$ +*)
 end

Under exact predicate match, neither the Stack *pop* nor the Queue *rest* specifications match with this query. Plug-in match does not work either because we need to assume Q_{pre} $(not(isEmpty(s)))$ to show $S_{post} \Rightarrow Q_{post}$. However, under generalized match, $Q5$ is matched by both of these. The proofs are very similar to that for $Q4$ in the weak post match (Figure 3.3).

Consider another example specifying a function that removes the most recently inserted element of a container. This query does not require that the specifier knows the axiomatization of containers, since the query uses only the container constructor, *insert*. The post-condition specifies that the input container, s, is the result of inserting the returned element, e, into another container ss. The existential quantifier (**there exists**) is a way of being able to name ss.

$\boxed{\text{Q6}}$

signature Q6 = **sig**
  (∗+ **using** Container +∗)
  type $\alpha\ T$ (∗+ **based on** Container.E Container.C +∗)
  val $delete : \alpha\ T \rightarrow \alpha$
  (∗+ $delete\ s = e$
    requires $not(isEmpty(s))$
    ensures there exists *ss:Container.C*
      $s = insert(e, ss)$ +∗)
 end

Again, under exact or plug-in matches, $Q6$ does not retrieve any functions. Under generalized match, the query retrieves the Stack *top* function, but not Queue *deq*, since the query specifies that the most recently inserted element is returned. To show $match_{gen}(Stack.top, Q6)$, we consider two cases: $s =empty$ and $s =insert(ec,sc)$. In the first case, the pre-condition for both *top* and *qTop* are false, and thus the match predicate is vacuously true. In the second case, the pre-conditions are both true, so we need to prove that $S_{post} \Rightarrow Q_{post}$. If we instantiate ss to sc, the proof goes through.

Specialized Match

For *specialized match*, we instantiate $\mathcal{R}$ in the generic predicate match to $\Leftarrow$. Specialized match is the converse of generalized match: $match_{spcl\text{-}pred}(S, Q) = match_{gen\text{-}pred}(Q, S)$. A function whose specification is weaker than the query might still be of interest as a base from which to implement the desired function. Specialized match allows the library specification to be weaker than the query.

Definition 3.2.9 (Specialized Match)

$$match_{spcl\text{-}pred}(S, Q) = Q_{pred} \Rightarrow S_{pred}$$

Consider again the query $Q3$, which is the same as Stack *top* but without the pre-condition. Stack *top* is thus weaker than $Q3$, but we can show that $Q3$ implies Stack *top* and hence that $Q3$ is matched by Stack *top* under specialized match.

3.3 Properties of the Matches

3.3.1 Equivalence and Partial Order

Name of Match	Predicate Symbol	Match Form	Kind of Match	$M_=$ (if Partial Order)
Exact Pre/Post	$match_{E\text{-}pre/post}$	Pre/Post	Equivalence	
Plug-in	$match_{plug\text{-}in}$	Pre/Post	Partial Order	Exact Pre/Post
Plug-in Post	$match_{plug\text{-}in\text{-}post}$	Pre/Post	Partial Order	$S_{post} \Leftrightarrow Q_{post}$
Weak Post	$match_{weak\text{-}post}$	Pre/Post	Neither	
Exact Predicate	$match_{E\text{-}pred}$	Predicate	Equivalence	
Generalized	$match_{gen\text{-}pred}$	Predicate	Partial Order	Exact Predicate
Specialized	$match_{spcl\text{-}pred}$	Predicate	Partial Order	Exact Predicate

Table 3.3: Summary of predicate symbol, match form, and kind of match for each function specification match.

In addition to distinguishing between instances of the generic pre/post matches and instances of the generic predicate matches, we classify the function specification matches by whether they are equivalence matches, partial order matches, or neither. Recall from Chapter 2 that both equivalence and partial order matches must be reflexive and transitive. In addition, equivalence matches are symmetric and partial order matches are antisymmetric. Of the seven matches defined in this chapter, only exact pre/post (Definition 3.2.3) and exact predicate (Definition 3.2.7) are equivalence matches. The other matches are all partial order matches except weak post match, which cannot be classified as either because it is not transitive. Partial order matches require an equivalence match ($M_=$) to prove they are antisymmetric. For generalized and specialized match, $M_= = match_{E\text{-}pred}$; for plug-in match $M_= = match_{E\text{-}pre/post}$; and plug-in post match uses the match $M_= = S_{post} \Leftrightarrow Q_{post}$ (essentially, dropping the pre-condition match in exact pre/post). Table 3.3 summarizes both classifications (Pre/Post or Predicate; Equivalence or Partial Order) and the equivalence match ($M_=$) used by each partial order match, as well as summarizing the names and predicate symbols for all the matches defined in this chapter.

Proving that the matches are equivalences or partial orders is straightforward and based on

1. Reflexive:
$$\begin{aligned} match_{plug\text{-}in}(S,S) &= (S_{pre} \Rightarrow S_{pre}) \ \wedge \ (S_{post} \Rightarrow S_{post}) \\ &= true \ \wedge \ true \\ &= true \ \Box \end{aligned}$$

2. Transitive:
$$\begin{aligned} \text{Given } (1)\ & match_{plug\text{-}in}(S,Q) = (Q_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow Q_{post}) \\ (2)\ & match_{plug\text{-}in}(Q,T) = (T_{pre} \Rightarrow Q_{pre}) \wedge (Q_{post} \Rightarrow T_{post}) \end{aligned}$$

Show $match_{plug\text{-}in}(S,T) = (T_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow T_{post})$

(a) Show $T_{pre} \Rightarrow S_{pre}$:

$$T_{pre} \stackrel{(2)}{\Rightarrow} Q_{pre} \stackrel{(1)}{\Rightarrow} S_{pre}$$

(b) Show $S_{post} \Rightarrow T_{post}$:

$$S_{post} \stackrel{(1)}{\Rightarrow} Q_{post} \stackrel{(2)}{\Rightarrow} T_{post} \ \Box$$

3. Antisymmetric:
$$\begin{aligned} \text{Given } (1)\ & match_{plug\text{-}in}(S,Q) = (Q_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow Q_{post}) \\ (2)\ & match_{plug\text{-}in}(Q,S) = (S_{pre} \Rightarrow Q_{pre}) \wedge (Q_{post} \Rightarrow S_{post}) \end{aligned}$$

Then $(Q_{pre} \Leftrightarrow S_{pre})$ and $(S_{post} \Leftrightarrow Q_{post})$

So $match_{E\text{-}pre/post}(S,Q)$ $\Box$

Figure 3.4: Properties of plug-in match.

the properties of $\Leftrightarrow$ and $\Rightarrow$. Consider, for example, plug-in match (Definition 3.2.4). Figure 3.4 shows the proof sketches of the reflexivity, transitivity, and antisymmetry of plug-in match.

3.3.2 Relating the Matches

We relate all the function specification match definitions in a lattice (Figure 3.5). An arrow from a match $M1$ to another match $M2$ indicates that $M1$ is stronger than $M2$ ($M1(S,Q) \Rightarrow M2(S,Q)$ for all S,Q). We also say that $M2$ is more relaxed than $M1$. The rightmost path in the lattice shows the pre/post matches; the remainder of the matches are predicate matches.

Table 3.4 summarizes which of the functions in the Toy Specification Library matches with each of the six example queries under each of the seven function specification matches we have defined. For example, under generalized match, $Q5$ is matched by both *Queue.rest* and *Stack.pop*, but under plug-in post match, $Q5$ is not matched by any functions in the library. Parentheses around a function indicates that the match is implied by a stronger match (e.g., $match_{plug\text{-}in}(Q2, Queue.enq) \Rightarrow match_{gen}(Q2, Queue.enq)$).

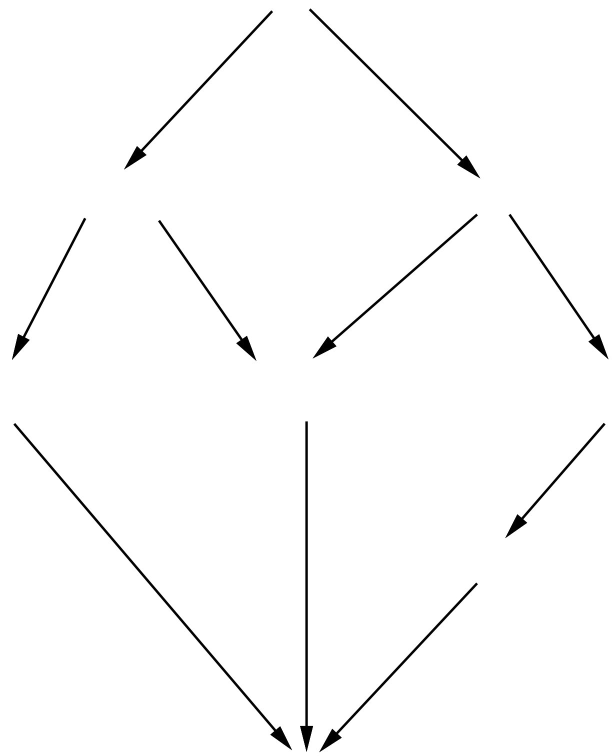


Figure 3.5: Lattice of function specification matches

	Exact Pre/Post	Exact Predicate	Plug-in	Specialized	Generalized	Plug-in Post	Weak Post
Q1	*Q.create* *S.create*	*(Q.create)* *(S.create)*	*(Q.create)* *(S.create)*	*(Q.create)* *(S.create)*	*(Q.create)* *(S.create)*	*(Q.create)* *(S.create)*	*(Q.create)* *(S.create)*
Q2	— —	— —	*Q.enq* *S.push*	— —	*(Q.enq)* *(S.push)*	*(Q.enq)* *(S.push)*	*(Q.enq)* *(S.push)*
Q3	—	—	—	*S.top*	—	*S.top*	*(S.top)*
Q4	— —	— —	— —	— —	— —	— —	*Q.rest* *S.pop*
Q5	— —	— —	— —	— —	*Q.rest* *S.pop*	— —	*Q.rest* *S.pop*
Q6	—	—	—	—	*S.top*	—	*S.top*

Table 3.4: Which functions match which queries ($Q =$ *Queue* module and $S =$ *Stack* module)

3.4 Implementation

We use LP, the Larch Prover [GG91], to attempt to prove that a match holds between two specifications. LP is a theorem prover for a subset of multisorted first-order logic. We have implemented tools to translate Larch/ML specifications and match predicates into LP input. Each of the specification match examples given in this chapter (i.e., all entries in Table 3.4) and in Section 5.3 have been specified in Larch/ML, translated automatically to LP input, and proven using LP.

For each specification file (e.g., `Stack.sig`), we check the syntax of the specification and then translate it into a form acceptable to LP. Namely, we generate a corresponding `.lp` file (e.g., `Stack.lp`), which includes the axioms from the appropriate LSL trait and contains the appropriate declarations of variables, operators, and assertions (axioms) for the pre- and post-conditions of each function specified. Each function *foo* generates two operators, *fooPre* and *fooPost*; the axioms for *fooPre* and *fooPost* are the bodies of the **requires** and **ensures** clauses of *foo*. Figure 3.6 shows `Stack.lp` and `Q2.lp`, the result of translating the Stack specification from the Toy Specification Library and the query $Q2$ into LP format. The **thaw** *Container_Axioms* command loads the state resulting from executing the commands in `Container_Axioms.lp`. We use the `lsl` tool to generate the file `Container_Axioms.lp` from the LSL trait `Container.lsl`. We comment out the **thaw** command in `Q2.lp`, since we assume that the query ($Q2$) uses the same trait as the library specification (*Stack*). The command **set name** $Q2$ tells LP to use $Q2$ as the prefix for names of facts and conjectures. Commands **declare var** and **declare op** declare variables and operators that will be used in the axioms. In particular, `Q2.lp` declares the element variable e, container variables $q1$ and $q2$, and operators *addPre* and *addPost*. The **assert** clause adds axioms to the logical system for *addPre* and *addPost*, corresponding to the **requires** and **ensures** clauses of *add*, respectively.

Given the names of two function specifications, their corresponding specification files, and which match definition to use, we also generate the appropriate LP input to initiate an attempt to show the match between those two functions. For example, Figure 3.7 shows the LP input to prove the plug-in match of Stack *push* with $Q2$. The input to LP for the proof consists simply of commands to load the theories for the library and query (**execute** *Stack* and **execute** $Q2$), and the proof statement (**prove** ...).

We could alternatively have chosen to generate the LP axioms on a per-query basis rather than generating axioms for each `.sig` file (i.e., given a particular pair of functions, generate only the necessary axioms for that particular pair). However, we assume that generating an `.lp` file from a `.sig` file will happen only once and that there may be several queries on a library specification or several match definitions for a particular query. This approach enables us to consider module-level matches as well.

Since LP is designed as a proof assistant, rather than an automatic theorem prover, some of the proofs require user assistance. Each of the 33 entries in Table 3.4 corresponds to a match

```
% Stack.lp
%% Using Container
thaw Container_Axioms
%% signature Stack
set name Stack
declare var
  e: E
  s: C
  s2: C
  ..

declare op
  createPre: ->Bool
  createPost: C ->Bool
  pushPre: ->Bool
  pushPost: C, E, C ->Bool
  popPre: C, C ->Bool
  popPost: C, C ->Bool
  topPre: C, E ->Bool
  topPost: C, E ->Bool
  ..

assert
  createPre = true;
  createPost(s) = (s = empty);
  pushPre = true;
  pushPost(s, e, s2) = (s2 = insert(e,s));
  popPre(s, s2) = (~(isEmpty(s)));
  popPost(s, s2) = (s2 = butLast(s));
  topPre(s, e) = (~(isEmpty(s)));
  topPost(s, e) = (e = last(s))
  ..
```

```
% Q2.lp
%% Using Container
%%% thaw Container_Axioms
%% signature Q2
set name Q2
declare var
  e: E
  q1: C
  q2: C
  ..

declare op
  addPre: C, E, C ->Bool
  addPost: C, E, C ->Bool
  ..

assert
  addPre(q1, e, q2) = (length(q1) < 50);
  addPost(q1, e, q2) =
    (length(q2) = length(q1) + 1)
  ..
```

Figure 3.6: LP input for *Stack* and *Q2*

that we have used LP to prove. In characterizing how much assistance the proofs require, we consider only the 14 entries in the table that are not in parentheses (call these the *primary matches*), since the proofs for entries in parentheses follow automatically from an entry to the left in the same row. Table 3.5 summarizes the level of user assistance required for the primary matches. *None* means the proof went through with no user assistance, *guidance* means that the proof required user input to apply the appropriate proof strategies, and *lemma* means that the user had to prove additional lemmas to complete the proof.

% PlugIn-Q2-Stack.lp
%% Load library and query specs
execute Stack
execute Q2

%% Plug-in Match: (Qpre => Spre) /\ (Spost => Qpost)
prove (addPre(s, e, s2) => pushPre) /\ (pushPost(s, e, s2) => addPost(s, e, s2))

Figure 3.7: LP input for plug-in match of *Stack.push* with $Q2$

Query	Library	Match	User Assistance
$Q1$	Queue.create	Exact Pre/Post	lemma
$Q1$	Stack.create	Exact Pre/Post	lemma
$Q2$	Queue.enq	Plug-in	none
$Q2$	Stack.push	Plug-in	none
$Q3$	Stack.top	Specialized	none
$Q3$	Stack.top	Plug-in Post	none
$Q4$	Queue.rest	Weak Post	lemma
$Q4$	Stack.pop	Weak Post	guidance
$Q5$	Queue.rest	Generalized	lemma
$Q5$	Stack.pop	Generalized	guidance
$Q5$	Queue.rest	Weak Post	lemma
$Q5$	Stack.pop	Weak Post	guidance
$Q6$	Stack.top	Generalized	guidance
$Q6$	Stack.top	Weak Post	guidance

Table 3.5: Level of user assistance required for LP proofs of queries

Four of the proofs needed no assistance from the user: plug-in match of *Stack.push* and *Queue.enq* with $Q2$, and plug-in post and specialized matches of *Stack.top* with $Q3$. Plug-in match of *Stack.push* with $Q2$ is the example shown in Figure 3.7; executing the statements in Figure 3.7 results in the response from LP that the match conjecture was proved by normalization; no user assistance was required.

Generalized match of *Stack.pop* with $Q6$ is an example of a match that requires some user assistance to LP. The user must tell the prover to use induction in the proof, and then how to instantiate the existential variables. Figure 3.8 shows an LP-annotated script for this proof. The lines with boldface are user input; <> and [ ] are proof notes from LP; and % is the comment character. The line `[ ] conjecture` indicates that LP completed the proof. We classify the user assistance for this proof as simply *guidance* – telling LP what proof strategy

```
% exec M-Gen-Q6-Stack
%% Load library and query specs
execute Stack
execute Q6

%% Generalized Match: (Spre => Spost) => (Qpre => Qpost)
prove (topPre(c, e) => topPost(c, e)) => (deletePre(c, e) => deletePost(c, e))
  % Additional user input
  resume by induction
    <> basis subgoal
    [ ] basis subgoal
    <> induction subgoal
    resume by specializing cc to cc1
      <> specialization subgoal
      [ ] specialization subgoal
    [ ] induction subgoal
  [ ] conjecture
  %% End of input from file 'Gen-Q6-Stack.lp'.
```

Figure 3.8: LP output for generalized match of *Stack.pop* with $Q6$

to use next in cases where the default strategies do not complete the proof. A total of five proofs require guidance: generalized and weak post matches of *Stack.top* with $Q6$, generalized and weak post matches of *Stack.pop* with $Q5$, and weak post match of *Stack.pop* with $Q4$.

The remainder of the proofs (exact pre/post match of *Queue.create* and *Stack.create* with $Q1$, weak post match of *Queue.rest* with $Q4$, and generalized and weak post matches of *Queue.rest* with $Q5$) required not only guidance but also additional *lemmas* in order to prove the match. In all five cases, one of the additional lemmas is *∼(insert(e,q) = empty)* (something that might reasonably be included in a more complete theory of containers). The proofs for *Queue.rest* with $Q4$ and $Q5$ additionally need the lemma *length(butFirst(insert(e,q))) = length(q)*, which falls out directly from the axioms for *Stack* but not *Queue*. The proofs for $Q1$ need additional lemmas about the lengths of containers. Figure 3.9 shows an LP-annotated script for the proof of weak post match of *Queue.rest* with $Q4$.

```
% exec M-Weak-Q4-Queue
%% Load library and query specs
execute Queue
execute Q4

set name Lemma
prove ~(insert(e,q) = empty) by contradiction
    <> contradiction subgoal
    critical-pair *Hyp with Container
    [ ] contradiction subgoal
  [ ] conjecture

prove length(butFirst(insert(e,q))) = length(q) by induction on q
    <> basis subgoal
    [ ] basis subgoal
    <> induction subgoal
    [ ] induction subgoal
  [ ] conjecture

set name Query
prove restPre(q, q2) /\ restPost(q, q2) => remainderPost(q, q2)
  resume by induction on q
    <> basis subgoal
    [ ] basis subgoal
    <> induction subgoal
    [ ] induction subgoal
  [ ] conjecture
  %% End of input from file 'Weak-Q4-Queue.lp'.
```

Figure 3.9: LP output for weak post match of *Queue.rest* with *Q4*

3.5 Discussion

In this section, we discuss the advantages and disadvantages of specification matching, why we chose to define the particular matches we did, and our implementation choices.

3.5.1 Specification Matching

Specifications provide a much richer description of a component, and thus specification matching is more discriminating than signature matching. For example, specification matching allows us to distinguish between functions for set union and set intersection, or between stacks and bags, which we cannot do with signature matching. If two component specifications match, we know that certain guarantees about the behavior of the system will hold if we substitute one component for the other. The particular guarantees depend on which match we use. Section 5.3 describes how to use specification matching to show substitution and subtyping relationships.

Specification matching is far more "expensive" in two respects: (1) the non-trivial overhead of writing the specifications and (2) the necessity of using theorem proving to show a match. Fortunately, we can use theorem proving technology to automate the match process rather than proving the match by hand. The additional overhead of specification matching is acceptable for applications where signatures are not descriptive enough and we are willing to expend a little extra effort to specify and prove that a match holds. For example, if we want to replace a component in a safety-critical system with an updated, verified library component, we would want to prove we could substitute the library component for the existing component.

3.5.2 Match Definitions

We define a variety of matches. As with signature matching, which match is most appropriate to use will depend on the particular situation. First, the choice of match depends on the context in which the match is used – how strong of a guarantee is needed about the relation between the two specifications? If we want to know that we can substitute one function for the other and still have the same behavior, we would use plug-in match or an exact match. In contrast, if we are only interested in whether the functions have the same effects and we are willing to check pre-conditions separately, we can use weak post match. Which match is most appropriate also depends on the actual form of the predicates. In some cases, pre/post matches will be easier to prove with a theorem prover since the pre/post matches relate pre-conditions to pre-conditions and post-conditions to post-conditions, and for two specifications, S and Q, it is likely that S_{pre} and Q_{pre} are related and hence we can reason about that relation (and similarly for S_{post} and Q_{post}). In other cases, however, it is necessary to make some assumptions about the pre-condition in order to prove a relation between the post-conditions. In these cases, the predicate matches are easier to prove.

There are, of course, many other variations of logical operators connecting $S_{pre}, S_{post}, Q_{pre},$

and Q_{post}, but they do not have the same correspondence to an intuitive relation between S and Q that the definitions presented here do. We should also note that not all of these other variations would fit into the two generic definitions we present in Section 3.2 since the generic definitions restrict the way terms are combined and which logical operators can be used. However, all would fit in the lattice of Figure 3.5 since the lattice has no such restrictions.

Another class of relaxed specification match definitions takes advantage of the structure of a particular predicate and relaxes the match in some way based on that structure. For example, if post-conditions were always in conjunctive normal form, we could define a relaxed match that allowed some of the conjuncts of either the library or the query to be dropped.

3.5.3 Choice of Language and Theorem Prover

There are two features of Larch/ML that affect the ease with which we can prove a specification match. The first feature is separate pre- and post-condition clauses. The matches in which we are interested are easy to express using this kind of specification, particularly the pre/post matches (Section 3.2.1). The pre/post matches best describe substitutability. We can substitute S for Q if S's pre-condition is weaker that Q's and S's post-condition is stronger than Q's (i.e., S works under at least as many conditions as Q and guarantees at least as much). We cannot express this notion without separating the pre- and post-conditions in the definition. Using a specification language with explicit pre- and post-conditions makes it easy to use pre/post matches. That does not preclude the use of specification matching on specification languages that do not explicitly separate the pre-condition, however. The predicate matches (Section 3.2.2) are still easy to use with such languages, and, with some additional work, we could also derive the pre- and post-conditions of the specifications to use the pre/post matches.

A second feature of Larch/ML is its use of the Larch two-tiered approach. We use the LSL tier to address the problem of needing a theory of common base terms when matching two specifications. In most specification languages (e.g., Z [Spi88], VDM [Jon86]), we would rely on the built-in expression language as the base theory. LSL makes it possible to define an extensible collection of general traits (e.g., *Container*), upon which Larch/ML specifications are built. The traits provide a common vocabulary for operators in two specifications that we want to match, each of which may be specified by different people. We assume that the number of such general traits would be relatively small, and thus specifiers could easily become familiar with them. We discuss how to prove a match between two specifications based on different traits in Section 7.2.2.

The disadvantage of the two-tiered approach is that it is, in some ways, twice the work. A user must learn not just one but two specification languages,[2] which adds to the already big effort required to use formal specifications at all.

[2]Even assuming that a set of general traits already exists, the user must learn enough about LSL to be able to use the traits.

Choosing Larch led naturally to using LP to prove specification matches. There is already a tool to translate LSL specifications to LP input, and implementing a translator for Larch/ML specifications was straightforward. LP is designed to be a proof assistant rather than a completely automatic theorem prover. Thus, it is not surprising that a number of our proofs required user guidance. Since our main concerns with the implementation of function specification match were to do some examples to demonstrate the validity of the approach and to compare the various match definitions, the need for user assistance was not a particular problem. However, it does raise questions about how hard it would be to prove match between significantly larger or more complex specifications.

Additionally, although our final proofs were not ultimately long or complex, we did spend quite a while to arrive at some of those proofs; theorem proving is like programming in that there is some "art" involved after all the theory and syntax are learned. An example of something that requires an artful touch in LP is the use of the *critical-pairs* command. Deciding when to apply the technique and to which pairs requires some experience. While this problem is not unique to specification matching, it is still worth noting that some of the same problems arise with specification matching as are found in other applications that use theorem provers.

Chapter 4

Module Matching

The previous two chapters address the problem of matching functions using signature and specification matching. However, a programmer may need to compare collections of functions, e.g., sets of operations on abstract data types. This chapter moves up a level in granularity to describe matching of modules.

Most modern programming languages explicitly support the definition of abstract data types through a separate modules facility, e.g., ML modules, CLU clusters, Ada packages, or C++ classes. Modules are also often used to group sets of related functions, like I/O routines. As with functions, providing the signature for a module is no additional work for the programmer, since either the programmer must provide the signature anyway for type checking, or the signature is generated automatically by type inference. Although specifications do require additional work, we hope that this thesis provides additional motivation for programmers to use formal specifications.

In this chapter, we define *module interfaces* (signatures and specifications at the module level) and *module matching*. At its core, a match between two modules requires a match between pairs of functions in the modules. The interesting thing about our module match definitions is that they have the function match as an explicit parameter. Thus, whether a module match is a signature match or a specification match depends only on whether the function match parameter is a function signature match (i.e., any match from Chapter 2) or a function specification match (i.e., any match from Chapter 3)

We begin the chapter with our definitions of module match (Section 4.1). As with the function match predicates, we define an exact match and various relaxations. Section 4.2 discusses the various properties of the matches – how they relate, which matches are equivalence or partial order matches, and various potential extensions to the match. Section 4.3 describes the implementation of module-signature-match-based retrieval, using the function signature match implementation from Section 2.5 and some shell scripts.

4.1 Match Definitions

A *module interface* is a pair, $\Sigma = \langle \Sigma_T, \Sigma_F \rangle$, where

- Σ_T is a set of user-defined types, and
- Σ_F is a set of function abstracts.

Σ_T introduces the names of user-defined type constructors that may appear in Σ_F. A function abstract is the function name together with either a function signature or a function specification. We include the function name both as useful feedback to the user and to distinguish between abstracts that would otherwise be the same (thus Σ_F is a set rather than a multiset). Within a given interface, all abstracts must be the same kind. If the abstracts are signatures, the interface is a *signature interface*; if the abstracts are specifications, the interface is a *specification interface*. We use τ to denote function abstracts. Since we can think of a signature as a very rudimentary specification, it is reasonable to use the same notation for both.

For the examples in this chapter, we use the Toy Signature Library in Chapter 1 as the library. This library contains three module signature interfaces: *List*, *Set*, and *Queue*. The *Set* interface has one user-defined type ($\Sigma_T = \{\alpha\ T\}$) and seven signatures in Σ_F. For consistency, we assume that the *List* interface has a type declaration for α *list*, even though *list* is a built-in type. Since our library for the examples consists of signature interfaces, we give examples of module signature matching. However, the definitions of module match apply for both signature and specification interfaces. Section 5.3.2 provides an example of module specification matching.

For a library interface, $\Sigma_L = \langle \Sigma_{LT}, \Sigma_{LF} \rangle$, to match a query interface, $\Sigma_Q = \langle \Sigma_{QT}, \Sigma_{QF} \rangle$, there must be correspondences both between Σ_{LT} and Σ_{QT} and between Σ_{LF} and Σ_{QF}. These correspondences vary for the exact and relaxed module matches.

4.1.1 Exact Match

Definition 4.1.1 (Exact Module Match)

$$
\begin{array}{l}
\textit{M-match}_E(\Sigma_L, \Sigma_Q, \textit{match}_{fn}) = \\
\quad \exists \text{ total functions} \\
\qquad U_F : \Sigma_{QF} \rightarrow \Sigma_{LF} \text{ and} \\
\qquad U_{TC} : \textit{UserOp}(\Sigma_{QT}) \rightarrow \textit{UserOp}(\Sigma_{LT}) \text{ (with corresponding renaming } TC) \\
\quad \text{such that (1) } U_{TC} \text{ and } U_F \text{ are one-to-one and onto} \\
\qquad\qquad (2)\ \forall\ \tau \in \Sigma_{QT}, \textit{match}_E(\tau, TC\ \tau) \\
\qquad\qquad (3)\ \forall\ \tau_q \in \Sigma_{QF}, \textit{match}_{fn}(U_F(\tau_q), TC\ \tau_q)
\end{array}
$$

U_{TC} and TC ensure that user-defined types are named consistently in the two interfaces. For a set of user-defined types Σ_T, $\textit{UserOp}(\Sigma_T)$ extracts the set of type constructor variables in

Σ_T (e.g., for $\Sigma_T = \{\alpha\ T, int\ X\}$, $UserOp(\Sigma_T) = \{T, X\}$). The domain of function U_{TC} is a set of type constructor variables; from it we construct the type constructor renaming sequence TC, which is applied to function signatures or to the signature part of function specifications. For each $u_q \in UserOp(\Sigma_T)$, the renaming $[U_{TC}(u_q)/u_q]$ appears in TC. To avoid potential naming conflicts, we assume that $UserOp(\Sigma_{QT})$ and $UserOp(\Sigma_{LT})$ are disjoint (if they are not, we can easily make them so).

U_F maps each query function abstract τ_q to a corresponding library function abstract, $U_F(\tau_q)$. Since any user-defined types in $U_F(\tau_q)$ come from Σ_{LT}, we apply TC to τ_q to ensure consistent naming of type constructors. The correspondence between each $TC\ \tau_q$ and $U_F(\tau_q)$ is that they satisfy the function match, $match_{fn}$. Since U_F is total, one-to-one, and onto, the number of functions in the two interfaces must be the same (i.e., $|\Sigma_{LF}| = |\Sigma_{QF}|$), and likewise for U_{TC}, Σ_{LT}, and Σ_{QT}.

The function match parameter ($match_{fn}$) gives us a great deal of flexibility, allowing any of the function matches defined in Chapters 2 or 3 to be used in matching the individual function abstracts in a module interface. For the matches to work, of course, the two interfaces and the function match must all be signatures (*module signature match*) or must all be specifications (*module specification match*).

Let us consider an example. Suppose we want a module that implements a functional abstract container. We describe it with the signature interface $M1$, shown below:

$$\begin{array}{lll} M1: & \Sigma_{QT} = \{ & \alpha\ C\ \} \\ & \Sigma_{QF} = \{ & create\text{: } unit \to \alpha\ C, \\ & & add\text{: } \alpha\ C * \alpha \to \alpha\ C, \\ & & delete\text{: } \alpha\ C \to \alpha, \\ & & remainder\text{: } \alpha\ C \to \alpha\ C\ \} \end{array}$$

If we use exact module match with exact function match, $M1$ is matched by the *Queue* interface in the Toy Signature Library (i.e., $M\text{-}match_E(Queue, M1, match_E)$) with $U_{TC}(T) = C$ and the obvious mapping of functions. Note that even allowing more relaxed function matches, exact module match of $M1$ with *List* or *Set* will never be true because both of these interfaces have more functions than $M1$, and U_F must be one-to-one and onto.

4.1.2 Partial Matches

Generalized Match

Should a querier really have to specify all the functions provided in a module in order to find the module? A more reasonable alternative is to allow the querier to specify only the functions of interest and match a module that is more *general* in the sense that its set of functions may properly contain the query's set.

Definition 4.1.2 (Generalized Module Match)

> $\textit{M-match}_{gen}(\Sigma_L, \Sigma_Q, \textit{match}_{fn})$ is the same as $\textit{M-match}_E(\Sigma_L, \Sigma_Q, \textit{match}_{fn})$ except U_{TC} and U_F need not be onto.

Thus, whereas with $\textit{M-match}_E(\Sigma_L, \Sigma_Q, \textit{match}_{fn})$, $|\Sigma_{LF}| = |\Sigma_{QF}|$, with $\textit{M-match}_{gen}(\Sigma_L, \Sigma_Q, \textit{match}_{fn})$, $|\Sigma_{LF}| \geq |\Sigma_{QF}|$, and $\Sigma_{LF} \supseteq TC\ \Sigma_{QF}$ (where $TC\ \Sigma_{QF}$ is a shorthand for applying TC to each element of Σ_{QF}).

In Standard ML, the notion of signature matching is applied in determining when a structure (code module) matches a signature. We can define this in terms of generalized module match. There are two conditions for a structure to match a signature. First, the structure must provide at least all the values (types and functions) declared in the signature. Second, the type of a function in the structure must be at least as general as that function's type in the signature. Let Σ_Q be an ML signature, and Σ_S be the actual signature of an ML structure S. Then the structure S matches the signature Σ_Q if $\textit{M-match}_{gen}(\Sigma_S, \Sigma_Q, \textit{match}_{gen})$.

Generalized module match is also the module match we expect to be most useful in practice. Library interfaces will typically provide a range of functions. A querier is likely to need only some of them and should have to specify as little as possible. Consider query $M1$ again, but with generalized module match. If we instantiate $\textit{match}_{fn}$ with $\textit{match}_E$, $M1$ is still only matched by *Queue*. However, if we use $\textit{match}_{tycon} \circ \textit{match}_{reorder}$ as the function match, $M1$ is matched by the *List* interface as well. However, it is not matched by *Set*, since *Set* does not have a function that matches with the *delete* or *remainder* functions in $M1$.

The following very simple query eliminates the distinction between modules that remove a specified object (like *Set*) and modules that remove a pre-determined object (like *List* and *Queue*):

$$
\begin{aligned}
M2\text{:}\quad \Sigma_{QT} &= \{\ \alpha\ C\ \} \\
\Sigma_{QF} &= \{\ \textit{create}\text{: } \textit{unit} \rightarrow \alpha\ C, \\
&\qquad \textit{add}\text{: } \alpha\ C * \alpha \rightarrow \alpha\ C\ \}
\end{aligned}
$$

$M2$ itself is a subset of $M1$ (i.e., $\textit{M-match}_{gen}(M1, M2, \textit{match}_E)$). In addition, with the function match $\textit{match}_{tycon} \circ \textit{match}_{reorder}$, $M2$ is matched by all three interfaces in the Toy Signature Library. Table 4.1 summarizes the examples, showing which library interfaces match with which queries under exact and generalized module match and various function matches.

Specialized Match

Specialized module match is the opposite of generalized module match. With specialized module match, a library need not have all the functions defined in the query. This is useful in practice.

Query	Module Match	$match_{fn}$: $match_E$	$match_{fn}$: $match_{reorder}$	$match_{fn}$: $match_{tycon} \circ match_{reorder}$
$M1$	Exact	*Queue*	*Queue*	*Queue*
$M1$	Generalized	*Queue*	*Queue*	*Queue* *List*
$M2$	Generalized	$M1$ *Queue*	$M1$ *Queue* *Set*	$M1$ *Queue* *Set* *List*

Table 4.1: Which modules match which queries

If a library module has most of the functionality you need, it may be possible to implement the remaining functions using the ones provided, or to use existing ones as prototypes.

Definition 4.1.3 (Specialized Module Match)

$$M\text{-}match_{spcl}(\Sigma_L, \Sigma_Q, match_{fn}) = M\text{-}match_{gen}(\Sigma_Q, \Sigma_L, \overline{match_{fn}})$$
$$\text{where } \overline{match_{fn}}(\tau_q, \tau_l) = match_{fn}(\tau_l, \tau_q)$$

$\overline{match_{fn}}$ reverses the order of the arguments to $match_{fn}$. In particular, $\overline{match_{gen}} = match_{spcl}$. We cannot simply use $M\text{-}match_{gen}(\Sigma_Q, \Sigma_L, match_{fn})$ because partial order matches are antisymmetric, so the order of arguments to $match_{fn}$ matters. For example, consider the following query $M3$:

$$\begin{aligned} M3\text{:} \quad \Sigma_{QT} &= \{\ string\ C\ \} \\ \Sigma_{QF} &= \{\ create\text{: } unit \to string\ C, \\ &\quad\ \ add\text{: } string\ C * string \to string\ C, \\ &\quad\ \ delete\text{: } string\ C \to string, \\ &\quad\ \ remainder\text{: } string\ C \to string\ C, \\ &\quad\ \ length\text{: } string \to int\ \} \end{aligned}$$

$M3$ is like *Queue* except that it uses a more specific type, *string C*, and has an extra function, *length*. Suppose we use the function match $match_{gen}$ with specialized module match. Then $M\text{-}match_{spcl}(Queue, M3, match_{gen}) = M\text{-}match_{gen}(M3, Queue, match_{spcl})$, which is true: the functions in *Queue* are matched by a subset of the functions in $M3$ under specialized function match. If we did not reverse the order of arguments to $match_{fn}$, the match would not hold (i.e., $M\text{-}match_{gen}(M3, Queue, match_{gen})$ is false) because the function signatures in $M3$ are not more general than those in *Queue*.

4.2 Properties of the Matches

4.2.1 Distinctions Between the Matches

The key difference between the three module match definitions lies in the relation between the sets of function abstract Σ_{QF} and Σ_{LF}. All three definitions use the same two mappings, U_{TC} and U_F, to draw correspondences between two interfaces; what varies is whether all functions in a particular interface are required to have a corresponding function in the other interface. That is, what varies is the relation between the sets Σ_{QF} and Σ_{LF}. Table 4.2 summarizes the set relation $\mathcal{R}$ for each match.

Match	$\Sigma_{QF}\ \mathcal{R}\ \Sigma_{LF}$
Exact	$=$
Generalized	$\subseteq$
Specialized	$\supseteq$

Table 4.2: Relation between Σ_{QF} and Σ_{LF} for the module matches.

4.2.2 Equivalence and Partial Order Matches

As with function matches, we classify module matches by whether they are equivalence matches or partial matches (Section 2.4.1, pg. 25). The classification is dependent to some extent on whether the function match used as a parameter to the module match ($match_{fn}$) is an equivalence match or a partial order match. Exact module match is an equivalence match when $match_{fn}$ is instantiated with an equivalence match, a partial order match when $match_{fn}$ is instantiated with a partial order match, and neither if $match_{fn}$ is neither. Because the subset relation is antisymmetric, generalized and specialized module match are partial order matches when $match_{fn}$ is either an equivalence match or a partial order match and neither if $match_{fn}$ is neither.

4.3 Implementation

We used Beagle, the retrieval tool for functions described in Section 2.5, and some shell scripts to implement *M-Beagle*, a signature-based retrieval tool for modules using generalized module match. We make two simplifying assumptions: (1) any type constructor renamings are handled by $match_{fn}$ (thus we ignore Σ_{QT} and Σ_{LT}) and (2) there are not two function signatures in Σ_{QF}

that match the same function signature in Σ_{LF}. Given a query interface $\Sigma_Q = \langle \Sigma_{QT}, \Sigma_{QF} \rangle$ (where $\Sigma_{QF} = \{\tau_1, \ldots \tau_n\}$) and a library of interfaces L, we do the following:

(1) For each $\tau_i \in \Sigma_{QF}$, create a set of modules $M_i \in L$.
$\Sigma_L \in M_i$ iff $\exists\ \tau_l \in \Sigma_{LF}$ such that $match_{fn}(\tau_l, \tau_i)$.

(2) $M = M_1 \cap \ldots \cap M_n$

M_i is the set of module interfaces that contain a function signature that matches τ_i. M is the intersection of the M_i's, and is the set of interfaces that match Σ_Q. That is, for $\Sigma_L \in L : \Sigma_L \in M$ iff $M\text{-}match_{gen}(\Sigma_L, \Sigma_Q, match_{fn})$.

If we eliminate our two simplifying assumptions, it would be necessary to write a slightly more complex tool that calculates TC and that tries several potential function mappings in the cases where multiple query function signatures match a library function signature (i.e., we would need to be able to try various permutations of the function mapping). In practice, such a tool should not be significantly more complex than our current implementation. We can reduce the search space for permutations with a few simple heuristics. One such heuristic would be to consider a mapping from τ_q to τ_l only if they have the same number of input arguments (if the function match is an uncurry match, do the uncurrying first). Another heuristic for the case where $match_{fn}$ does not include $match_{gen}$ or $match_{spcl}$ would be to check for occurrences of the same base types in τ_q and τ_l.

In the case where we simply wish to compare a particular pair of interfaces (e.g., for subtyping), it is also reasonable to require the user to supply the mappings U_{TC} and U_F. For example, for specification interfaces, if the user supplies a pairing of function names, we automatically generate the LP assertions for the match (we assume U_{TC} is the identity function). See Section 5.3.2 for details.

4.4 Discussion

The module matches are highly parameterized and extensible. The function match relation between the pairs of functions is completely orthogonal to the module match definitions; we can instantiate $match_{fn}$ with any of the signature or specification function matches. In fact, we could easily define another relaxed module match where the function match can vary on a per-function-abstract basis.

Most generally, a module interface consists of some global information (Σ_T) and a set of functions (Σ_F). This framework allows the potential to extend the module interface to contain even more information. One such instance is the way the definitions allow not only signatures but also specifications. Additionally, we could extend module specification interfaces to include information about shared types or global invariants in Σ_T. A new module match definition including global invariants would be similar to Definition 4.1.1 of exact module match, but

U_{TC} would change and point (2) of the definition would require some kind of consistency between invariants.

We are able to use module matching to relate two modules. Section 5.3.2 contains an example of how we use module specification match to show that one type is a behavioral subtype of another. When comparing two modules to show they are related (e.g., one is a subtype of another), the focus is on the relations between the types and functions provided by the modules, i.e., their interfaces, which is exactly the focus of these module matches.

In cases where we want to locate a set of functions with particular types, module match can be used for retrieval, but it is less effective for this. One problem is that whereas with function matching, it is very easy for a user to "tweak" their query and relaxations based on previous results, the tweaking in a module must be done on a per-function basis, which increases the number of queries exponentially. Consider a simple example. Suppose we need to create a container and use the query $unit \rightarrow \alpha\ C$. Suppose that does not retrieve a satisfactory result, so we use the query $\alpha\ C$ to try to find initial values for empty containers. Next, suppose we want to find a function to add an element to a container. We are not sure whether we need an additional argument for the location of the element in the container, so we try both $\alpha * \alpha\ C \rightarrow \alpha\ C$ and $\alpha * \beta * \alpha\ C \rightarrow \alpha\ C$. In both these cases, there is very little overhead involved in trying two queries rather than one. But if this were a module match, we would have to try all four combinations of create and insert signatures in order to cover all the possibilities.

The problem here is that the focus of the module matches is the interface of a component (the types and functions provided by the module), but the conceptual goal of someone who wants to retrieve a component is likely to be more abstract, for example, "I need a container" rather than "I need a component that has a create function and an add function." So what we need is a more abstract notion of the type of a module, in addition to the interface. With such a module type system, we could then use function signature matching to search for what we want, for example, for a program that takes a file in *dvi* format and converts it to Postscript, or we could search for a container or a parser, in a type system that contains those things as basic types.

Chapter 5

Applications

This chapter describes applications of both signature and specification matching. We discuss three main kinds of applications: retrieval (Section 5.1), indexing (Section 5.2), and substitution (Section 5.3). Retrieval applications return the subset of components in a library that match a query by the user. These subsets are useful in locating components for reuse or in analyzing, browsing, or filtering the library. Indexing applications define an index on the library for efficient storage and retrieval of components and for browsing the library. Substitution applications compare two components using a signature or specification match. Depending on the match, we can guarantee various properties will hold when we substitute one component for the other. For each application, we illustrate with examples from signature or specification matching as appropriate, using the implementations described in Sections 2.5, 3.4, and 4.3.

5.1 Retrieval

One of the central problems with software libraries is the need to search for and retrieve components from the library. At the heart of any solution to the retrieval problem is some way of comparing a description of what is desired from the library (the query) with each component in the library, to see whether it matches. We use abstracts (signatures or specifications) to describe the components, and any of the signature or specification matches to do the comparison. Formally, we define the retrieval problem as follows:

Definition 5.1.1 (Retrieve)

$$\begin{aligned} \mathit{Retrieve}\colon\ & (\text{Component} * \text{Set of Components} * (\text{Component} * \text{Component} \rightarrow \text{Bool})) \\ & \rightarrow \text{Set of Components} \end{aligned}$$

$$\mathit{Retrieve}(Q, L, M) = \{C \in L : M(C, Q)\}$$

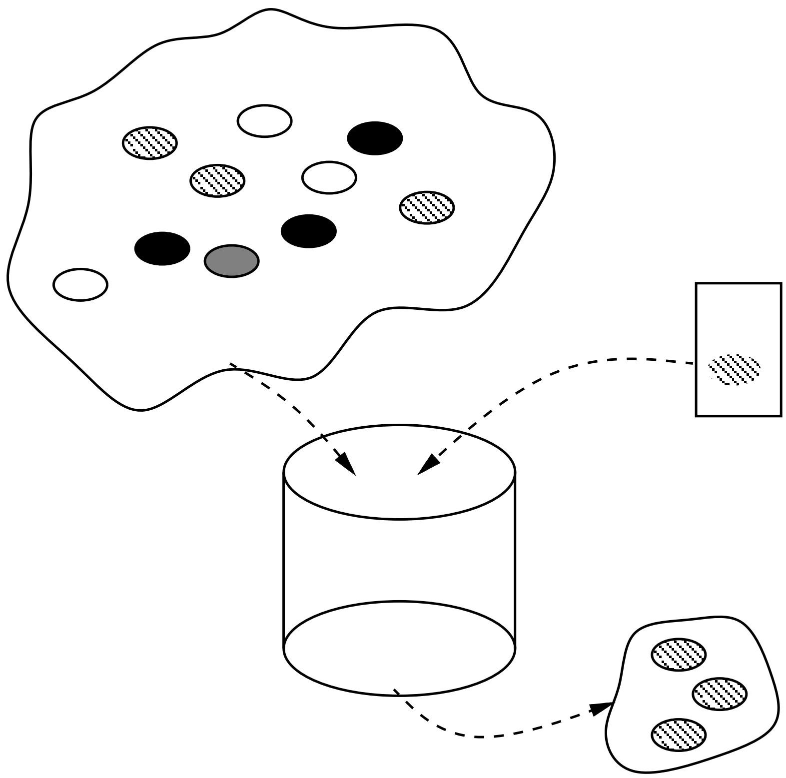


Figure 5.1: The retrieval problem

Figure 5.1 illustrates this definition. Given a query component Q, a signature or specification match M, and a library (set of components) L, *Retrieve* returns the set of components in L that match with Q under M. The components may be either functions or modules and components may contain either signature or specification abstracts (or both), provided that M is instantiated with an appropriate match. Parameterizing the definition by M also gives the user the flexibility to choose the degree of relaxation in the match. Retrieval is *signature-based* when M is instantiated with a signature match, and *specification-based* when M is instantiated with a specification match.

There are many reasons we might want to describe or retrieve a subset of library components. These include

- To locate a particular component for reuse
- To analyze the library
- To browse the library
- To use the resulting subset as the library for another retrieval or to combine it with the result of another retrieval (i.e., to do compound retrieval)

The following examples are drawn from actual use of the signature-based retrieval tools Beagle and M-Beagle on the Community Library by ourselves and our colleagues. They illustrate the usefulness of allowing the user to specify which relaxations to use for a match, as well as showing successful use of signature-based retrieval for reuse, for analysis, for browsing, and in compound retrieval. In the reuse examples, we also include explanations of some cases where functions matched but were not what we wanted, since they help us understand more about how retrieval works in practice.

5.1.1 Reuse

The most obvious and widely discussed application of retrieval is to locate components for reuse. Components may be reused directly or may need to be modified slightly.

Reuse 1

As part of the implementation of a version of Beagle, we needed to generate a list of "tag bits" (all initialized to false) to track which elements of a list have already been used. Thus, we needed a function that takes a boolean b and an integer n and generates a list of length n where each element has value b. Since it seemed likely that a library function would be more general in the list's element type we used the query $(\alpha * int) \rightarrow \alpha\ list$ with relaxations reorder and uncurry on an earlier implementation of Beagle. This search results in exactly one match, the function *create* (with type $int \rightarrow \alpha \rightarrow \alpha\ list$, from the *List* module in the Edinburgh sub-library), which does exactly what we want. If we do not take the step of generalizing from *bool* to α on our own, but instead use the query $(bool * int) \rightarrow bool\ list$ with relaxations reorder, uncurry, and generalized, we retrieve 28 functions instead of just *create*, since any of *bool*, *int*, *bool list*, and the tuple $bool * int$ can be generalized. Of the 28 functions retrieved by the more general match, 24 have type $\alpha \rightarrow \beta$, 3 have type $\alpha \rightarrow \beta \rightarrow \gamma$, and 1 (*create*) has type $int \rightarrow \alpha \rightarrow \alpha\ list$. This second query illustrates the tradeoffs in using relaxed matches, such as generalized match and specialized match, that instantiate type variables: increasing recall may reduce precision, so the query may retrieve more useless components.

Reuse 2

A colleague of ours needed a function to take two lists and create a list of pairs of elements from those lists. He used the query $\alpha\ list \rightarrow \beta\ list \rightarrow (\alpha * \beta)\ list$ with relaxation uncurry, which retrieves three functions: the *zip* function in all three sub-libraries. The code for all three functions is the same (except that the *zip* from the CMU sub-library is curried while those in the other two libraries are uncurried), and all three do exactly what he wanted.

Reuse 3

In another case, we needed to convert the representation of a type constructor name from a list of strings to a single string with the elements of the list separated by "."s (e.g., convert ["Parser", "Table", "T"] to "Parser.Table.T"). We used the query $\mathit{string}\ \mathit{list} \rightarrow \mathit{string}$ with no relaxations. Notice that in this case we do not want to allow generalization, since we are implicitly assuming that the function will use string concatenation, which would not generalize. This query retrieves six functions, including *pathImplode* and *implodePath*, from the SML/NJ and CMU sub-libraries respectively. Both of these functions take a list of strings and returns a string which is the concatenation of those strings with "/"s between the strings (to form a path name) and both are easy to modify to do what we want by replacing "/"s with "."s. The other four functions out of the six retrieved include *implode* from the Edinburgh sub-library, which does not put a separator between the strings, and three functions that perform other manipulations on a list of strings that form a directory pathname.

Another reasonable query to use for this example is one that also includes the separator string as a parameter. The query $(\mathit{string}\ \mathit{list} * \mathit{string}) \rightarrow \mathit{string}$ with relaxations reorder and uncurry retrieves only the function *firstLine* (in both the *execute* and *getwd* modules in the CMU sub-library), which does something completely different (it takes a program name and list of arguments as input, and returns the first line of program output as its output string). This example shows that a relaxation to allow more or fewer arguments in a tuple would be useful.

A third approach to this example is to assume that there is a more general function that takes as input a function on strings in addition to the list of strings. One might try a query like $\mathit{string}\ \mathit{list} \rightarrow ((\mathit{string} * \mathit{string}) \rightarrow \mathit{string}) \rightarrow \mathit{string}$. This query with just reorder and uncurry relaxations does not match anything. Adding the generalized relaxation results in 30 matches, the majority of which have type $\alpha \rightarrow \beta$ or $\alpha \rightarrow \beta \rightarrow \gamma$ and are not useful. Two of the results, *foldL'* and *foldR'* (from the Edinburgh sub-library), have type $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha\ \mathit{list} \rightarrow \alpha$. The function *foldL'* is like the built-in function *fold* except that *foldL'* uses the first element of the list as the initial value, whereas *fold* requires an initial value as an additional input (*foldR'* is similarly related to the built-in function *revfold*). We write a function, *specialConcat*, to concatenate two strings with a "." in between, and call *foldR'* with *specialConcat* and the list of strings to achieve our goal.

Reuse 4

An example that gives a somewhat surprising result is the query $\alpha\ \mathit{list} \rightarrow \alpha$ using specialized match. We might expect this query to find functions that return an element from the list, like *hd* (although built-in functions are not in the library, so *hd* itself is not returned). This query results in nine matches, six of which have type $\mathit{string}\ \mathit{list} \rightarrow \mathit{string}$ (the same ones described in Reuse 3). Of the other three functions, two are *flatten* functions from different sub-libraries (type α *list list* $\rightarrow \alpha$ *list*), and the only one like *hd* is *last* from the Edinburgh sub-library.

Reuse 5 (Module Reuse)

Retrieval for reuse also occurs at the module level. Recall the module signature query $M2$ presented in Chapter 4. The query defines an abstract container type and functions to create and add to a container.

$$\begin{aligned} M2: \quad \Sigma_{QT} &= \{\ \alpha\ C\ \} \\ \Sigma_{QF} &= \{\ \mathit{create}\colon\ \mathit{unit} \rightarrow \alpha\ C, \\ &\qquad \mathit{add}\colon\ \alpha\ C * \alpha \rightarrow \alpha\ C\ \} \end{aligned}$$

Using M-Beagle and the function signature match with type constructor, reorder, and uncurry relaxations, $M2$ retrieves two modules: the *sortableQueue* module (with functions *empty* and *enq*) and the *lstream* (lazy stream) module (with functions *empty* and *cons*), both from the CMU sub-library.

A more common way to initialize a container is to define a constant that is the value of the empty container (e.g., [ ] on lists). $M3$ shows a module signature query for this approach.

$$\begin{aligned} \text{M3:} \quad \Sigma_{QT} &= \{\ \alpha\ C\ \} \\ \Sigma_{QF} &= \{\mathit{create}\colon\ \alpha\ C, \\ &\qquad \mathit{add}\colon\ \alpha\ C * \alpha \rightarrow \alpha\ C\ \} \end{aligned}$$

Using the same match instantiations as for $M2$, $M3$ retrieves four modules for sets, queues, and lists from the Community Library. Table 5.1 summarizes details of the four modules, showing the name of the module, the names of the constants and functions matching the function signatures, and which sub-library the module is in. The two queries, $M2$ and $M3$, find most of the immutable containers in the library. They do not, however, retrieve the *IntSet* and *BinarySet* modules from the SML/NJ sub-library because the set type in these modules does not use a polymorphic variable. For example, the *BinarySet* module contains the value *empty : set* and the function $add : set * item \rightarrow set$.

Even with just two functions in the query module, using module signatures rather than function signatures significantly reduces the number of matches. Using function-signature-based retrieval with the same relaxations (type constructor, reorder, and uncurry), the query $unit \rightarrow \alpha\ C$ ($M2.create$) retrieves twelve functions, the query $\alpha\ C$ ($M3.create$) retrieves nine functions, and the query $\alpha\ C * \alpha \rightarrow \alpha\ C$ (add) retrieves nine functions.

Module	$\alpha\ C$	$\alpha\ C * \alpha \to \alpha\ C$	Sub-library
EqSet	*empty*	*insert*	Edinburgh
fifo	*empty*	*enqueue*	SML/NJ
fifo2	*empty*	*enq*	CMU
List	*empty*	*updateLast*	Edinburgh

Table 5.1: Results of module library query $M3$

5.1.2 Statistical Analysis

Another use of retrieval is to analyze a software library. We can characterize properties of the library, make general statements about the types of the functions in the library, or generate a set of components we then further analyze by hand. Consider the following examples.

Analysis 1

We can use retrieval to gather statistical information about the size of a library. For example, to find the total number of functions in the library, we use the query $\alpha \to \beta$ with specialized match. When applied to the Community Library, this query retrieves 1451 functions. Note that the query α with specialized match is matched by constants as well as functions. If we use α as the query with specialized match (i.e., count both constants and functions), we find 1739 components in the Community Library.

Similarly, we can use module match to count the number of modules in a library. The module query consists of just one function of type $\alpha \to \beta$ and the match uses generalized module match with specialized function match. Using this query, M-Beagle finds 129 modules in the Community Library.

Analysis 2

A fellow graduate student wanted to gather statistics about what percentage of functions in the libraries have a curried form (i.e., return a function). Under specialized match, the query $\alpha \to \beta \to \gamma$ retrieves 577 functions out of 1451 (40%). Using the same query, we observe a distinction among the sub-libraries: 51% of the functions in the Edinburgh sub-library have this form (352 out of 688), compared with 27% of the SML/NJ sub-library (98 out of 362) and 32% of the CMU sub-library (127 out of 401).

To find out more about the form of curried functions, we use the queries $q_4 = \alpha \to \beta \to \gamma \to \delta$, $q_5 = \alpha \to \beta \to \gamma \to \delta \to \epsilon$, $q_6 = \alpha \to \beta \to \gamma \to \delta \to \epsilon \to \zeta$, and $q_7 = \alpha \to \beta \to \gamma \to \delta \to \epsilon \to \zeta \to \eta$. Using these queries with relaxation specialized retrieves 184, 32, 7, and 0 functions, respectively.

Each query $q_{n+1} = v_1 \to \ldots \to v_n \to v_{n+1}$ retrieves a subset of the functions retrieved by the query $q_n = v_1 \to \ldots \to v_n$, since q_{n+1} is itself a specialization of q_n (by instantiating v_n in q_n to $v_n \to v_{n+1}$). Thus, when we reach q_7, which retrieves no functions, we know that the longest curried functions in the library are instantiations of q_6. We use this fact in the next analysis.

Analysis 3

We wanted to see how the various base types (*bool*, *int*, *real*, *string*, and *unit*) are used in the library. For each base type, *bt*, we used the query $\alpha \to bt$ with relaxations specialized, uncurry, and reorder to find out which functions return a value of type *bt*.

To find out how many functions take a type *bt* as one of the inputs is more difficult because there is no single query that can match functions with an arbitrary number of inputs, one of which is type *bt*. However, the number of inputs to a function is finite and usually fairly small. We know from Analysis 2 that the uncurried versions of curried functions in the library have no more than five elements in the input tuple. A scan of the library types with `grep` shows that the largest tuple in the library has four elements. Thus, we use queries of the form $bt * v_1 * \ldots * v_{n-1} \to v_n$, where the v_i's are type variables and n ranges from two to five. We use the relaxations specialized, uncurry, and reorder; specialized match allows instantiation of the v_i's, uncurry allows us to also count curried functions, and reorder allows *bt* to occur anywhere in the tuple. We must use the uncurry relaxation on the query $bt \to \alpha$ to avoid matching functions of the form $bt \to X \to Y$, since these are counted in the matches for the query $bt * \alpha \to \beta$.

Table 5.2 shows the results of the sequence of queries to find usage of the various base types and for a variable. Each query was run with relaxations specialized, reorder, and uncurry. Each entry shows the number of functions retrieved by that query, where *bt* in the query is instantiated by the base type shown at the top of each column. For example, for the base type *real* (fourth column), the library contains 23 functions that return a real number (i.e., the query $\alpha \to real$ with relaxations specialized, uncurry, and reorder retrieves 23 functions). There are 15 functions that take a real number as their only input; 24 functions that either take a real

Query	*bool*	*int*	*real*	*string*	*unit*	variable
$\alpha \to bt$	197	81	23	164	198	1451
$bt \to \alpha$	3	41	15	132	87	1451
$bt * \alpha \to \beta$	9	159	24	169	1	629
$bt * \alpha * \beta \to \gamma$	1	55	1	38	0	184
$bt * \alpha * \beta * \gamma \to \delta$	2	17	0	15	0	34
$bt * \alpha * \beta * \gamma * \delta \to \epsilon$	0	2	0	5	0	7

Table 5.2: Usage of various base types (*bt* stands for the base type used in each column)

number as part of an input pair or are curried functions of the form $X \rightarrow Y \rightarrow Z$, where X or Y is *real*; and 1 function that either takes a real number as part of its three-element input tuple or has an equivalent curried form. Thus, a total of 40 functions (the sum of all entries in the column but the first) have the type *real* as one of the inputs. The rightmost column in Table 5.2 uses a type variable rather than a base type for *bt* in the queries, and thus answers more general questions about the number of input elements to functions in the library. For example, the query $\delta * \alpha * \beta \rightarrow \gamma$ retrieves 184 functions, each of which has a three-element input tuple (or an equivalent curried form).

Alternatively, we could have used something like `grep` to count the total number of functions that use a particular base type, but we could not have broken down the results based on the number of arguments or whether the base type is part of the input or part of the output.

Analysis 4

We can also use retrieval to find out how many functions have n elements in the input tuple, for n ranging from two to four (we know we can stop at four from the results of Analysis 3). For each n, we use the query $v_1 * \ldots * v_n \rightarrow v_{n+1}$ and specialized match. Table 5.3 shows the results of the queries. These results are the number of matches with the queries using only the specialized relaxation. For each query q_n, the number of functions retrieved by q_n in Table 5.3 is less than the number of functions retrieved by q_n in Table 5.2 because we do not include the uncurry relaxation here, since we do not want to include curried functions in these counts.

Query	Number of matches
$\alpha * \beta \rightarrow \gamma$	256
$\alpha * \beta * \gamma \rightarrow \delta$	32
$\alpha * \beta * \gamma * \delta \rightarrow \epsilon$	9

Table 5.3: Number of functions with input tuples of various sizes

5.1.3 Retrieval-based Browsing

Although having a structure on top of the library to browse through is generally preferable (see Section 5.2), retrieval is also useful for browsing. Retrieval provides a way to "break up" the library into more manageable pieces, or focus on a subset of functions that are likely to be of interest. Browsing through what has been retrieved allows a user to learn about the style of a particular programming language or library, find out what is available in the library, see examples of how to program using a particular data structure or kind of function, or simply become more familiar with the contents of a library. In fact, using retrieval to browse a library can also help a user become familiar with a retrieval tool as well, by simply performing queries with various relaxations and browsing through the results to see what matched.

Browsing 1

We wanted to see whether naming conventions for user-defined types in ML were the same for each sub-library. Most user-defined types are likely to have at least one function that returns an object of that type, so we used the query $\alpha \to \beta\ X$ with relaxations specialized and type constructor. Browsing through the results of using this query on the SML/NJ and Edinburgh sub-libraries, we found that user-defined types tend to have meaningful names and to be lowercase (e.g., *array*, *fifo*, *splay*, and *intmap* in the SML/NJ sub-library, and *array* and *vector* in the Edinburgh sub-library). We also found some inconsistencies in naming. For example, in the SML/NJ sub-library, the hash table type is *hash_table* while the integer map type is *intmap* (i.e., one compound word uses an underscore and the other does not); in the Edinburgh sub-library, some of the user-defined types are lowercase (*array* and *vector*) while others are not (*Set* and *Const*). Looking through the results of using the same query on the CMU sub-library, we noticed that many of the functions name the user-defined type constructor T. This is not because they are the meant to be the same type, but rather simply a naming convention adopted by many ML programmers.

Browsing 2

We can also use browsing to learn something about how to use a particular data structure. For example, to learn how to use higher-order functions, we begin by looking at the results of Analysis 2, which shows the percentage of higher-order functions for each of the sub-libraries. The Edinburgh sub-library has the highest percentage of higher-order functions, so we browse through the results of the query $\alpha \to \beta \to \gamma$ with specialized match on the Edinburgh sub-library to find examples of higher-order functions. To really see the power of making a function like *nth* ($int \to \alpha\ list \to \alpha$) higher-order, we would also have to find uses of it (e.g., a function $second{:}\alpha\ list \to \alpha = nth\ 1$). We also find functions such as *iterate* ($int \to (\alpha \to \alpha) \to \alpha \to \alpha$) and *exists* ($(\alpha \to bool) \to \alpha\ list \to bool$), which take a function as a parameter and apply the function either a certain number of times (*iterate*) or to each element of a list (*exists*), another common use of higher-order functions.

Browsing 3

Suppose we are interested in seeing how side-effecting functions work. These functions often return *unit* since the actual work in the function is in modifying an object rather than creating a new value. The query $\alpha \to unit$ with specialized match retrieves 114 functions from the Community Library. Browsing through some of these, we find a few general kinds of functions: I/O functions that close or write to an I/O stream, print functions, operating system functions (e.g., *pwd*, *cd*), thread control flow, and updates on array-like data structures. We could browse specific functions to see more details of how to implement each of these kinds of functions.

Browsing 4

When motivating the weak post specification match (Definition 3.2.6, pg. 46), we gave examples of functions that match the signature but not the specification of the query $Q4$ (shown again below).

signature Q4 = **sig** $\boxed{\text{Q4}}$
  (*+ **using** Container +*)
  type $\alpha\ T$ (*+ **based on** Container.E Container.C +*)
  val *remainder* : $\alpha\ T \to \alpha\ T$
  (*+ *remainder s* = *s*2
    ensures $length(s2) = (length(s) - 1)$ +*)
 end

To find these examples, we use the query $\alpha\ T \to \alpha\ T$ with relaxation type constructor to retrieve 13 functions that match with $Q4$'s signature. We then characterize the functions by browsing through them, and looking at the code if the function name does not clearly indicate the general purpose of the function. We classify the retrieved functions as follows:

- Four functions return the container that results from removing an element from the input container (i.e., they informally match with $Q4$ under weak post match). Those functions are *dropLast* and *tl* (on lists), *tail* (on lazy streams), and *deq* (on queues).

- Four other functions also return a subpart of the input container, but do not necessarily decrease the size of the container by one (and hence do not match with $Q4$). The *dropRepeats* function on lists removes duplicate elements from the list; the *rootptr* function returns the root of a mergeable reference (which is itself a mergeable reference) – there are two different versions of this function in the library; and the *copy* function returns a full copy of a hash table.

- Three functions reverse an ordered container: *rev* on lists, vectors, and arrays.

- The remaining two functions also permute or transform the container in some way. The *lrotate* function rotates splay trees, and the *mkblk* function marks a red-black tree as black.

All of the functions in the last three groups are examples of functions that match the signature but not the specification of $Q4$.

5.1.4 Compound Retrieval

The result of retrieval is a set of components. *Compound retrieval* combines the results of two retrievals using set operations (intersection, union, set difference). If $R1 = \mathit{Retrieve}(Q1, L, M1)$ and $R2 = \mathit{Retrieve}(Q2, L, M2)$, then

- $R1 \cup R2 = \{C \in L : (M1(C, Q1) \vee M2(C, Q2))\}$

  The union is the set of components in L that match with $Q1$ under $M1$ or match with $Q2$ under $M2$.

- $R1 - R2 = \{C \in L : M1(C, Q1) \wedge \neg M2(C, Q2)\}$

  The set difference is the set of components in L that match with $Q1$ under $M1$ but do not match with $Q2$ under $M2$.

- $R1 \cap R2 = \{C \in L : M1(C, Q1) \wedge M2(C, Q2)\}$

  The intersection is the set of components in L that match both with $Q1$ under $M1$ and with $Q2$ under $M2$.

The library (L) need not even be the same in the two retrievals, but must have the same granularity (i.e., functions or modules). $M1$ and $M2$ may be matches that require different retrieval tools (e.g., $M1$ is a signature match and $M2$ is a specification match), providing that the components in the library include both kinds of abstracts.

In the case where two retrievals use the same library and the set operation between them is $\cap$ or $-$, we can *pipeline* the retrievals. We form a pipeline by using the result of the first retrieval, $R1$, as the library of the second retrieval. Pipelining increases the efficiency of a compound retrieval by reducing the number of components that must be checked by $M2$. This is particularly useful if $M1$ is a faster match than $M2$; we use $M1$ to "weed out" obvious non-matches before applying $M2$. For example, $M1$ might be a signature match and $M2$ a specification match.

The result of a pipelined pair of retrievals is equivalent to the intersection of the retrievals if both are done on the original library. That is, if $R1 = \mathit{Retrieve}(Q1, L, M1)$, $R2 = \mathit{Retrieve}(Q2, L, M2)$, and $R2' = \mathit{Retrieve}(Q2, R1, M2)$ (i.e., the pipelined result), then $R2' = R1 \cap R2$. Further, the results of set difference of $R1$ with $R2$ and of $R1$ with $R2'$ are equivalent (i.e., $R1 - R2 = R1 - R2'$).

Figure 5.2 illustrates how pipelining works. The boxes represent the *Retrieve* function. Arrows entering the box represent inputs to the function (the query, components, and match), and the arrow leaving the box represents the output (i.e., the result of the retrieval). Figure 5.2.a illustrates the general notion, where $R1 = \mathit{Retrieve}(Q1, L, M1)$ and $R2' = \mathit{Retrieve}(Q2, R1, M2)$. Figure 5.2.b shows an example of pipelining a signature match and a specification match. The first retrieval uses the function signature query $\alpha\ T \rightarrow \alpha$ and exact function signature match on the Toy Specification Library with the result $R1 = \{\mathit{Stack.top}, \mathit{Queue.deq}\}$. The second

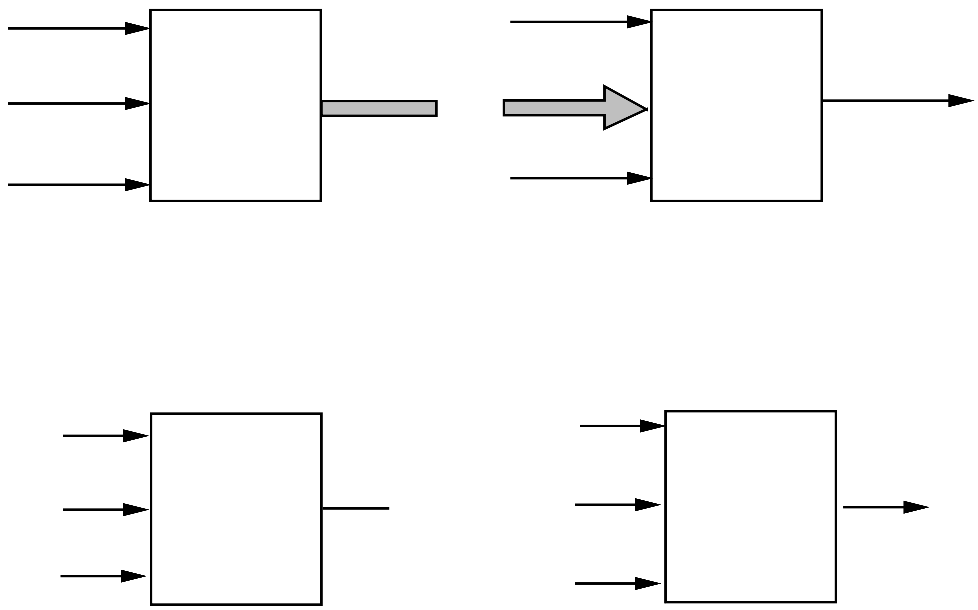


Figure 5.2: The idea behind pipelining

retrieval then uses the specification query $Q6$ from Section 3.2.2 (pg. 49; $Q6$ specifies that the function returns the most recently inserted element of the input container), and generalized predicate match to retrieve *Stack.top* from $R1$. The advantage to using pipelining here is that we only had to apply the specification match to the two functions in $R1$ rather than to all eight functions in the Toy Specification Library.

We now consider examples of compound retrieval using set union, difference, and intersection. All examples in this section were obtained by using Beagle on the Community Library and doing the set operations by editing the search results and using various Unix tools (e.g., `diff`, `cat`).

Compound 1 (using $\cup$)

We use set union to answer disjunctive queries. Suppose we want to look at functions that return a path name of a file, but are unsure whether the path name is represented as a string or a list of strings where each element in the list is a directory in the path. To find all functions that return either a string or a list of strings, we cannot use just one query. Instead, we use two separate signature-based retrievals and combine the results, as shown in the following sequence of queries and library manipulations:

1. Let $R1$ be the result of the query $\alpha \rightarrow string$ on the library with specialized match, i.e., $R1 = Retrieve(\alpha \rightarrow string, Community, match_{spcl})$.

2. Let $R2$ be the result of the query $\alpha \rightarrow string\ list$ on the library with specialized match, i.e., $R2 = Retrieve(\alpha \rightarrow string\ list, Community, match_{spcl})$.

3. Let $R3 = R1 \cup R2$. $R3$ contains the functions that return either a string or a list of strings.

When we perform these queries, $R1$ contains 105 functions, $R2$ contains 22 functions, and thus, $R3$ contains 127 functions that return either a string or a list of strings. For example, in $R1$ we find the function *getwd: unit* $\rightarrow$ *string*, from the *cshellDir* module in the CMU sub-library, which returns the current working directory (as a string). In $R2$ we find the function *clearPath': string list* $\rightarrow$ *string list*, from the *pathname* module in the CMU sub-library, which processes the ".." and "." components of a path to create a "clear" path (as a list of strings). Additionally, we find the functions *pathImplode* (in $R1$) and *pathExplode* (in $R2$), which convert between the list of string and string formats for path names.

Compound 2 (using $-$)

We use set difference to find the components that match one query but not another. Suppose we want to find functions that take two inputs, of which one is a real number and the other is not. We use the query $real * \alpha \rightarrow \beta$ with relaxations specialized, uncurry and reorder to find functions that take a real number as one of its inputs, but this may include functions with real numbers as both inputs. We then use a second query, $real * real \rightarrow \beta$, again with relaxations specialized, uncurry and reorder to find and filter out those functions that have two real numbers as inputs, as we show in the following sequence of queries and library manipulations:

1. Let $R1$ be the result of the query $real * \alpha \rightarrow \beta$ on the library, L, with specialized match, i.e., $R1 = Retrieve(real * \alpha \rightarrow \beta, Community, match_{spcl})$.

2. Let $R2$ be the result of the query $real * real \rightarrow \beta$ on $R1$ with specialized match, i.e., $R2 = Retrieve(real * real \rightarrow \beta, Community, match_{spcl})$.

3. Let $R3 = R1 - R2$. $R3$ contains the functions that take two inputs, of which one is a real number and the other is not.

As previously noted, if we use L rather than $R1$ to create $R2$, the result of $R3$ is still the same, but using $R1$ is more efficient. The first query retrieves 24 functions and the second query retrieves 13. Thus, $R3$ contains 11 functions with a real number as only one of its two inputs, for example *$**$* $: (real * int) \rightarrow real$ and $mkRandom : real \rightarrow (unit \rightarrow real)$.

Compound 3 (using $-$)

Suppose we want to find functions of the form $real \to X$, where X is not a function type. The query $real \to \alpha$ with specialized match will retrieve functions of that form, but may also retrieve functions of the form $real \to Y \to Z$, where Y and Z are type expressions, since α could be instantiated with a function type. We filter out the functions of the form $real \to Y \to Z$ with the following sequence of queries and library manipulations:

1. Let $R1$ be the result of the query $real \to \alpha$ on the library with specialized match, i.e., $(R1 = Retrieve(real \to \alpha, Community, match_{spcl})$.

2. Let $R2$ be the result of the query $real \to \alpha \to \beta$ on $R1$ with specialized match, i.e., $(R2 = Retrieve(real \to \alpha \to \beta, R1, match_{spcl})$.

3. Let $R3 = R1 - R2$. $R3$ contains the functions of the form $real \to X$, where X is not a function type.

When we perform these queries, $R1$ contains 25 functions, and $R2$ contains 10, leaving us with 15 functions in $R3$. Ten of those have type $real \to real$, four have type $real \to int$, and one has type $real \to (real * real)$. An alternative way to get this same result is to use a single query with the uncurry relaxation in addition to specialized, since this will first uncurry functions of the form $real \to Y \to Z$ to $(real * Y) \to Z$, and hence will not match the query.

Compound 4 (Using $\cap$)

We use set intersection to do conjunctive queries, even when we use different retrieval tools for different parts of the query. Suppose we want a function that returns the length or size of an object. We could look for functions with signature $\alpha \to int$ using signature-based retrieval, or for the string "length" in the text of the code using `grep`, or we can look for things that match both queries. Because `grep` does not distinguish between different functions in a module (file), we do both queries at the module level. Also, `grep` searches the full text, so "length" may be function name, may be a called function, or may occur in the comments.

1. Let $R1$ be the result of the query $\alpha \to int$ on the module library with specialized match, i.e.,. $R1 = Retrieve(\alpha \to int, Community, match_{spcl})$.

2. Let $R2$ be the result of a string search for the word "length" in modules in the library, i.e., $R2 =$ `grep length *.sml`.

3. Let $R3 = R1 \cap R2$.

These queries result in 36 modules in $R1$, 28 modules in $R2$, and 10 modules in $R3$. Nine of the modules contain functions that do what we want, named either *length* (three functions), *size* (five functions), or *len* (one function). The tenth module contains the function *skipBlanks*, which returns an integer position in a string (after skipping blanks) and uses string length in its calculations.

Whereas $R1$ and $R2$ both have many modules that do not match our intent, $R3$ contains only one "wrong" match. Thus, using conjunctive queries can increase the precision of the match.

Compound 5 (Using $\cap$)

The implementation of M-Beagle, the module-signature-based retrieval tool described in Section 4.3, is also an example of compound function retrieval. Assume the following:

- $Retrieve'(Q, L, M)$:
  (Function $*$ Set of Modules $*$ (Function $*$ Function $\rightarrow$ Bool)) $\rightarrow$ Set of Modules
  $Retrieve'(Q, L, M) = \{C \in L : \exists\ \tau \in C : M(\tau, Q)\}$
- $M_{mod}(Q, L) = \textit{M-match}_{gen}(Q, L, M)$ (where M is a function match)
- $\Sigma_Q = \langle \Sigma_{QT}, \Sigma_{QF} \rangle$ where $\Sigma_{QF} = \{\tau_1, \ldots, \tau_n\}$

Then we define module retrieval as follows:

$$\textit{M-Retrieve}(Q, L, M_{mod}) = \bigcap_{i=1}^{n} Retrieve'(\tau_i, L, M)$$

$Retrieve'(Q, L, M)$ is a modified function-based retrieval that returns the set of modules in L that contain functions that match with Q under M. Module retrieval ($\textit{M-Retrieve}(Q, L, M_{mod})$) is then a compound retrieval using set intersection on the results of $Retrieve'$ for each function in Q.

5.1.5 Discussion

The examples in this section demonstrate how signature-based retrieval is used to locate functions for reuse, statistical analysis, and browsing. Most of the examples were actual uses of Beagle by ourselves and our colleagues. We were able to use Beagle to retrieve the functions we wanted. In fact, for some cases (e.g., the statistical analysis examples), the queries could not have been answered without a signature matcher.

Retrieval for reuse is useful if it is fast and easy to retrieve, select, and reuse a component. We consider each factor (retrieval, selection, and reuse) in turn to see how using function signatures helps the reuse process.

First, signature-based retrieval is fast and easy. Function signatures are easy to write, and retrieval using Beagle is reasonably fast. The search is linear in the size of the library, or better if we use an indexed library. For libraries of thousands or tens of thousands of functions, we can expect search times of a few seconds or less. (The average search time for the Community Library of 1451 functions is .13 seconds.)

Whether or not it is easy to select an appropriate component from those retrieved by a query depends upon the contents of the library and the kind of query. The library should be large enough so that there is a high likelihood of finding something useful (and also too large for random browsing to be effective). For queries that use generalized match, it is particularly bad to have a large number of functions with very general signatures like $\alpha \to \beta$ or $\alpha \to \beta \to \gamma$. The Edinburgh library, for example, has 24 functions of type $\alpha \to \beta$, including functions on streams and functions for systems calls like *cd* or *pwd*. These functions will match any function query that uses the generalized relaxation, and yet they are not likely to be appropriate for most queries, so a library with a lot of these types of functions will have poor precision for any queries using generalized match. Queries for certain kinds of functions are particularly good for signature-based retrieval. The include queries for functions that are data oriented, such as functions on abstract data types and iterators (like *fold* and *map* on lists).

The third factor affecting the usefulness of retrieval for reuse is how easy it is to reuse a component once it is found. There are three general ways that a component can be reused. First, it may be reused exactly. For example, using the *zip* function to create a list of pairs of functions (Reuse 2). A function requiring no modification is very easy to reuse by simply calling the function (and linking the appropriate library, if necessary). Second, it may be possible to modify a retrieved function slightly and then reuse it. For example, modifying the *pathImplode* function to separate strings with a "." rather than a "/" (Reuse 3). Third, it may be possible to use a very general program, perhaps by writing a smaller "helper" function. For example, using *foldR'* with the helper function *specialConcat* to concatenate a list of strings (Reuse 3). This last class of appropriate functions are ones that are not likely to be found by string-based retrieval, since they are usually very general functions.

Even for relatively small libraries, such as the set of built-in functions for a language, signature-based retrieval is useful for finding functions whose names are not known or for helping a programmer learn a new language. For example, a functional programmer who is familiar with SML knows that the function *fold* applies a function to a list accumulating a result (type $(\alpha * \beta \to \beta) \to \alpha\ list \to \beta \to \beta)$, but this function is called *reduce* in Hope [FH88], *it_list* in the CAML Light core library [Ler95a], and *fold_left* in the CAML Special Light standard library [Ler95b].

As another example consider Lisp, which has over 500 functions listed in the manual index [Ste84]. A signature-based retrieval tool would have been useful to have had when we used Lisp to implement the gnu-emacs interface for Beagle, even though gnu-emacs has a good keyword search facility (`apropos`) and we had a reasonable manual. To find the function to write a string to an output file, for example, we could use `apropos` on "file" or "string" and get too many functions (93 for "file" and 36 for "string"), or we could look up "file" or "string" in the manual index, and again be overwhelmed. What we really wanted was to be able to ask for the functions that take a string and a file as input, i.e., signature match with the query $string * file \rightarrow unit$. It turns out there is not such a function, but we could have spent far less time discovering that fact.

Signatures vs. Text

Signature-based retrieval provides some advantages over text-based retrieval. First, signature-based retrieval enables us to do statistical analysis on the type of library functions, which we could not do with text-based retrieval. Second, using generalized match, we can find more general functions that can be instantiated to satisfy the query. The names of these more general functions are unlikely to be related to any keywords used by a text-based retrieval. For example, in the third approach to Reuse 3, keywords for a function with type $string\ list \rightarrow ((string * string) \rightarrow string) \rightarrow string$ might be "implode," "compress," or "convert," but are unlikely to be "fold" or any of the other keywords from the documentation that would retrieve the *foldL'* and *foldR'* functions. A third advantage of signature-based retrieval is that, unlike text-based retrieval, we know that any function retrieved by a signature query (using any relaxations but specialized), we are able to transform the type of the retrieved function into the query type. This makes it easier to use the function and could allow us to automate the reuse of a library function.

For cases where these advantages of signature-based retrieval are not important, other factors in deciding whether to use signature-based retrieval or text-based retrieval include how easy it is to express a query for either method and how easy the tools are to use. In many cases, a programmer initially thinks of a function in terms of its parameters. Thus, it is most natural to express the query in terms of the signature; there is no need to "translate" from thinking in terms of the programming language to thinking in terms of descriptive words for text-based retrieval. In some cases, there is an obvious word to express the desired function, such as "sort" or "create," and text-based retrieval is thus most appropriate. In other cases, there are several possible words for a function, such as "add," "insert," and "enqueue," or there is no obvious word to use for text-based search, but the type of the function is easy to describe, so signature-based retrieval is most appropriate. For those unfamiliar with certain programming terminology, the function names "zip" (Reuse 2), "implode" (Reuse 3), or "fold" (Reuse 3) would not have been at all obvious, but we successfully used signature-based retrieval to find the desired functions. Thus, the particular problem and how the user is thinking about it influences which method has an easier or more obvious query. Because signature-based retrieval is the better choice in many cases, it should be as easy to use as text-based retrieval techniques like the search commands in emacs or the Unix `grep` utility. Therefore, signature-based retrieval should be well-integrated with the programming environment so that a user may choose the appropriate approach based on the easiest way to express a particular query.

Limitations

Signature-based retrieval is not always the best tool for the job. In some cases, there is not enough information in the signature to discriminate between a large number of functions. For example, many numerical operations have the types $int * int \rightarrow int$ or $real * real \rightarrow real$, and many side-effecting functions have the general type $\alpha \rightarrow unit$. For these kinds of queries, signature matching will return a lot of matches between which the user will then have to discriminate further. Specification matching is one way of helping to discriminate further between functions with the same signatures.

Another problem with signature matching arises when using types that contain user-defined type constructors. If we know, for example, that queues have the type $\alpha\ Q$, then we can use that in our queries. But if we are not sure of the name of the user-defined type, we must use the type constructor relaxations to allow our name for the user-defined type to match with all other user-defined types. This assures us that we do not miss anything with a different type constructor name, but may result in matches with functions we are not interested in as well.

Additionally, as we discussed in Section 4.4, signature-based retrieval of modules is not effective in cases where the intent of the query is at a higher level of abstraction than the module interface.

Which Relaxations are Best

Function signature-based retrieval illustrates the advantages of allowing various combinations of relaxations, since the intended use of the retrieved component affects which relaxations will be used. In the case of retrieval for reuse, there are three relaxations or combinations of relaxations that we expect to be used most frequently (if none are used, then we are using exact match):

- Reorder and Uncurry. Use these if the format of the input does not matter. These are likely to be used together if they are used at all.

- Type Constructor. Use this if the names of user-defined types are unknown.

- Generalized. Use this if a more general function might also be useful. Generalized match may greatly increase the number of functions retrieved, often diluting the interesting hits, so it should be used with care.

In contrast, retrieval for statistical analysis and for browsing uses type variables as query variables. Hence, we expect to use specialized match (the only relaxation we do not expect to be especially useful for reuse). Whether a statistical analysis or browsing query uses any of the other relaxations (reorder, uncurry, or type constructor) will vary depending on the particular query, based on the same guidelines described above.

The relaxations used for compound retrieval will depend on whether the ultimate goal of the retrieval is reuse, statistical analysis, or browsing.

5.2 Indexing

Another class of applications uses signature or specification match definitions to create an index for a component library. An index for a library is analogous to an index for a book or map in that it is a structure over the library. We create an *indexed library* from a component library by organizing the components into a graph. Each node in the graph contains a set of *equivalent* components, and an edge from one node to another indicates that the first node is *more general* than the second. We use an *index pair*, which is a pair of signature or specification matches, to determine equivalence between components and to determine whether one component is more general than another.

We use the additional information provided by an indexed library for efficient storage and retrieval of components and for navigation when browsing a library. To show how indexed libraries aid in these applications, we assume for now that we have function components and use signature matches to form the indexed library, where exact match (Definition 2.2.2) defines nodes, which contain equivalent components, and generalized match (Definition 2.2.6) defines edges, which indicate that one component is more general than another. We present the formal definition of an indexed library in Section 5.2.1.

Storage and Retrieval

Consider first the storage of components. A user interested in one component in a node is very likely also to be interested in equivalent components (which would be in the same node of an indexed library). Storing components of a node together can thus improve locality of references and could also be useful in predictive fetching. Depending on the application, a user may also be interested in the parents or children of a given node and thus smart storage of related nodes could also improve performance.

Second, consider the use of indexes for retrieval from a library. Given a query, Q, we need only check whether Q is matched by the node signature rather than by each component signature, since all components at a node have equivalent signatures. Thus, we reduce the number of match comparisons required to retrieve the components that match with Q.

We can use the hierarchical structure of the index to prune the retrieval search space. For example, let Q be a query signature and suppose the retrieval match is exact match. We traverse the indexed library, beginning with the most general node, to retrieve all components that match with Q. Once we find a node whose signature matches with Q, we are done, since all components whose signatures match with Q must be in that node. Additionally, if we find a node, N, that Q is more general than, we prune the children of N, since none of them can be more general than Q. By pruning in this way, we further reduce the number of match comparisons required for a retrieval.

Structure-based Browsing

The structure of an indexed library also provides a natural framework for browsing through a library. Given a particular function, a user could request the next most general functions, or the functions that are equivalent modulo tuple reordering. For example, if users were looking at the *intsort* function from the Toy Signature Library in Figure 1.2 (pg. 6), they might want to know if there is a more general sorting function that works for arbitrary types of list elements. With an indexed library, they could simply ask to see all functions whose types are more general than *intsort*.

For function signature match, an index defines a type hierarchy. If we apply the same approach to module signature or specification matching, using the signature or specification notion of subtyping (Section 5.3.2), we have a class hierarchy. Thus, some aspects of object browsers like the Smalltalk browser [Tes81] can be viewed as an instance of our more general notion of browsing on an indexed library.

5.2.1 Indexed Library Definition

Rather than view the index as a separate structure, we define an *indexed library* – a single structure that contains both the index and the contents of the library. We use the terms index and indexed library interchangeably throughout this section.

An indexed library is a directed acyclic graph. Nodes represent equivalent components; edges order components based on their relative generality. The notions of equivalence and generality are precisely defined by a pair of match predicates, $(M_=, M_>)$, called the *index pair*. More formally:

Definition 5.2.1 (Indexed Library)

MakeIndex : Library $*$ Index Pair $\rightarrow$ Indexed Library
MakeIndex$(L, (M_=, M_>)) = \widehat{L}_{(M_=, M_>)}$

Given a library L and an index pair $(M_=, M_>)$, the indexed library $\widehat{L}_{(M_=, M_>)}$ is a directed acyclic graph.

Nodes:

Each node is an equivalence class defined by $M_=$.

Each node $n \in \widehat{L}_{(M_=, M_>)}$ has two parts:

- *n.sig* – a function signature.
- *n.elements* – a list of library components whose signatures are equivalent (under $M_=$) to *n.sig*.

Edges:

The edges form a partial order over L.

In order for indexing to work properly, the index pair $(M_=, M_>)$ must be instantiated according to the following restrictions:

1. $M_=$ must be an equivalence match.

2. $M_>$ must be a partial order match.

3. $M_=$ must be the corresponding equivalence match for $M_>$, so that

$$M_>(c1, c2) \wedge M_>(c2, c1) \Leftrightarrow M_=(c1, c2)$$

4. There must be a maximal node, $maxNode$, such that $\forall n \in \widehat{L}_{(M_=, M_>)}$, $M_>(maxNode, n)$.

Equivalence match, partial order match, and the corresponding equivalence match are defined in Section 2.4.1 (pg. 25). The third restriction ensures that $M_=$ and $M_>$ will work together properly. The partial order matches with which $M_>$ is instantiated are not strict; they include a notion of equality (as defined by the corresponding equivalence match of $M_>$). We use $M_=$ to "weed out" equal components; hence, $M_=$ and $M_>$ must have the same notion of equality. Thus, the partial order relation defined by the edges of an index is strict, and we use $M_>$ rather than $M_\geq$. The maximal node ($maxNode$) is the root of the graph and serves as the starting point for any traversals of the index.

For example, if $M_=$ is $match_{tycon}$, then $M_>$ must be $match_{tycon} \circ match_{gen}$ ($match_{spcl}$ would not guarantee us a maximal node). In general, for any equivalence match $M_=$ on function signatures, the corresponding $M_>$ is $M_= \circ match_{gen}$ and the maximal node has the signature α (all other types are instantiations of the type variable α).

We define indexed libraries for the case of function signature matching, so the index is a type hierarchy. Replacing *n.sig* with a module signature or a function or module specification and using appropriate index pairs would define indexes for module signatures or for function or module specifications. Using module matching with a definition of subtyping (Section 5.3.2) creates an index that is a subtype hierarchy.

Figure 5.3 illustrates an indexed version of the Toy Signature Library from Figure 1.2 (pg. 6) using index pair $((match_{tycon} \circ match_{reorder} \circ match_{uncurry}), (match_{tycon} \circ match_{reorder} \circ match_{uncurry} \circ match_{gen}))$. Each shaded rectangle is a node, n, with *n.sig* in the upper white rectangle and *n.elements* in the lower white rectangle. For example, the lower rightmost node has $n.sig$ = $unit \rightarrow \alpha\ T$ and $n.elements$ = (*Set.create*, *Queue.create*, and *List.empty*). *Set.create*:$unit \rightarrow \alpha\ T$ and *List.empty*:$unit \rightarrow \alpha\ list$ are both elements because $M_=$ on this index allows type constructor renaming. An arrow from a node $n1$ to another node $n2$ indicates that $M_>(n1, n2)$. For example, there is an arrow from the second node in the second column to the lower rightmost node because $M_>(\alpha \rightarrow \beta\ T, unit \rightarrow \alpha\ T)$.

Figure 5.3 includes some "special" nodes with the single element, *Special*: the maximal node of type α, and two others with types $\alpha\ T \rightarrow \beta$ and $\alpha \rightarrow \beta\ T$ to show more of a hierarchy. In

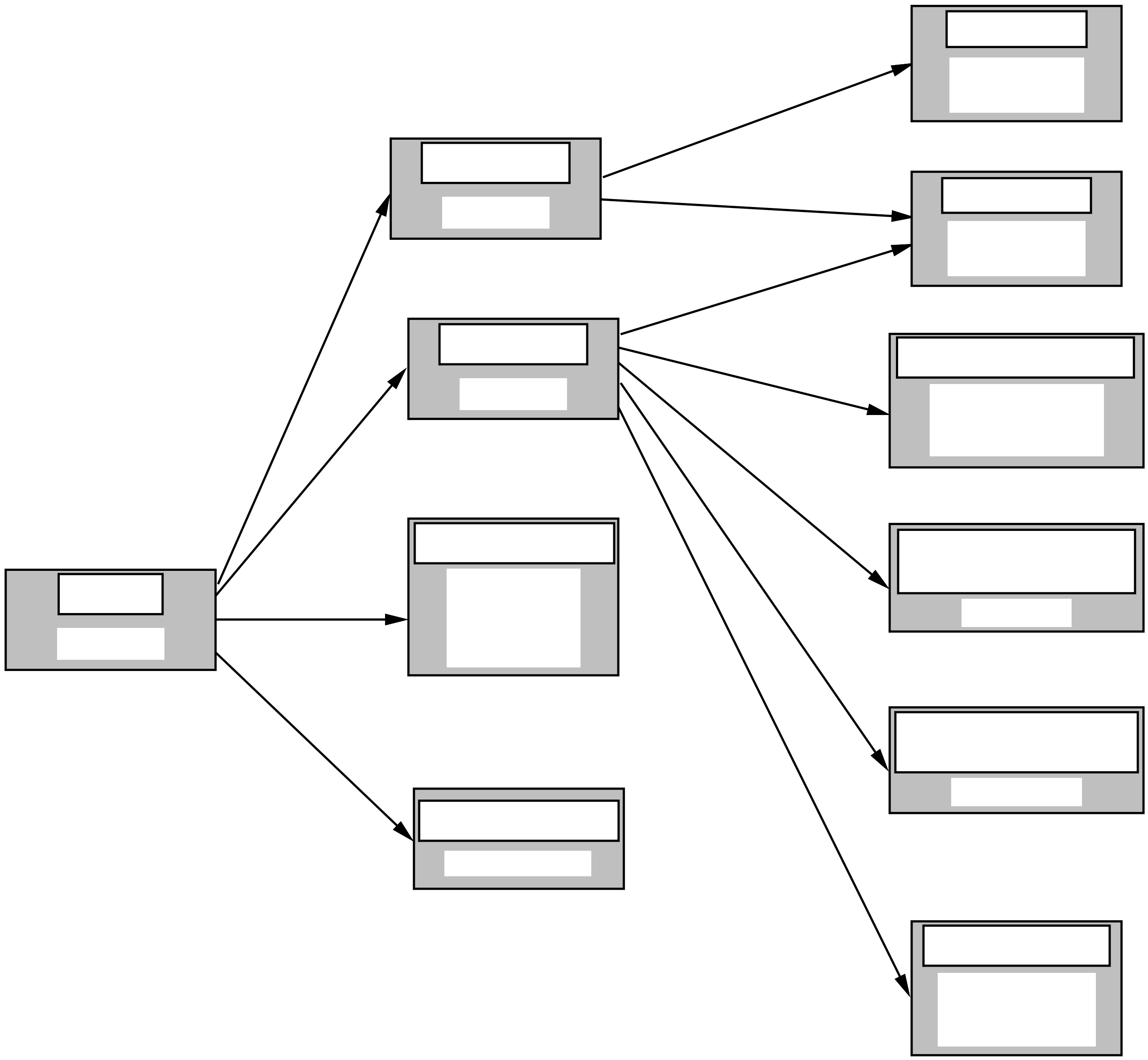


Figure 5.3: Indexed library for the Toy Signature Library with added "special" nodes. Index pair = $(match_{tycon} \circ match_{reorder} \circ match_{uncurry}, match_{tycon} \circ match_{reorder} \circ match_{uncurry} \circ match_{gen})$.

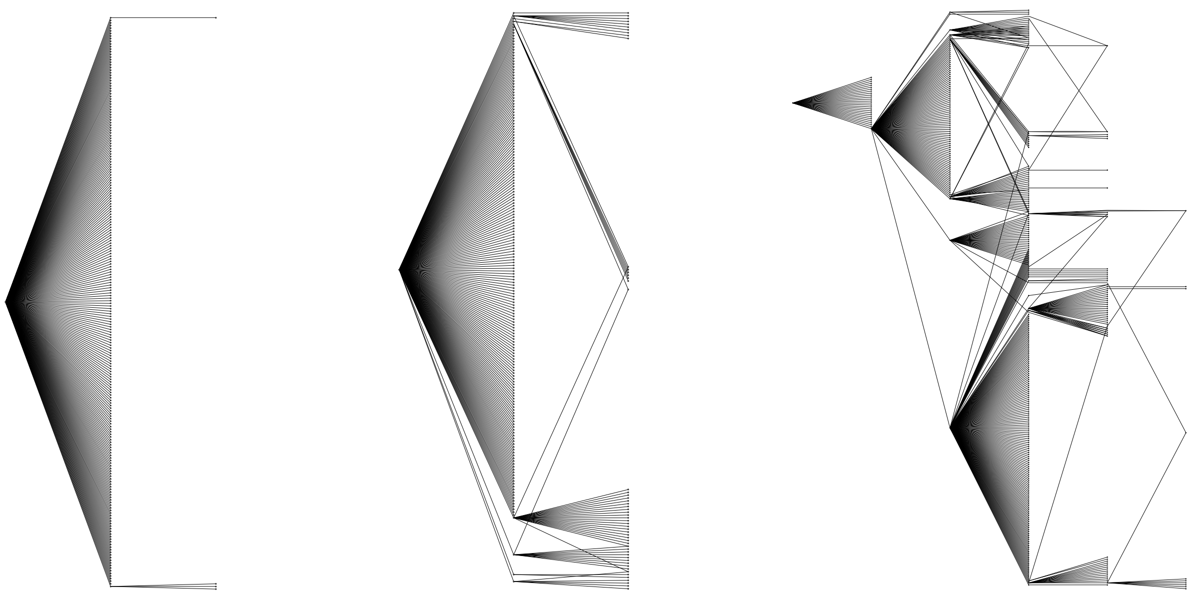
the case of a library that does not have many polymorphic types, this is a way of adding more depth. This structure has three levels: the maximal node α at the first level; four nodes on the second level (the two special nodes and two of the component nodes); and six of the component nodes at the third level. The eight component nodes comprise the seventeen components in the library, since in many cases, one node contains multiple components.

5.2.2 Indexes on the Community Library

We implemented a *MakeIndex* function and then built and analyzed indexes for the Community Library and each of its sub-libraries using two different index pairs:

1. $EQ = (match_E,\ match_{gen})$
2. $R\text{-}U = (match_{reorder} \circ match_{uncurry},\ match_{reorder} \circ match_{uncurry} \circ match_{gen})$.

The *EQ* index pair is the strictest possible match; it allows only exact matches in equivalence classes. The *R-U* index pair allows reordering and uncurrying.

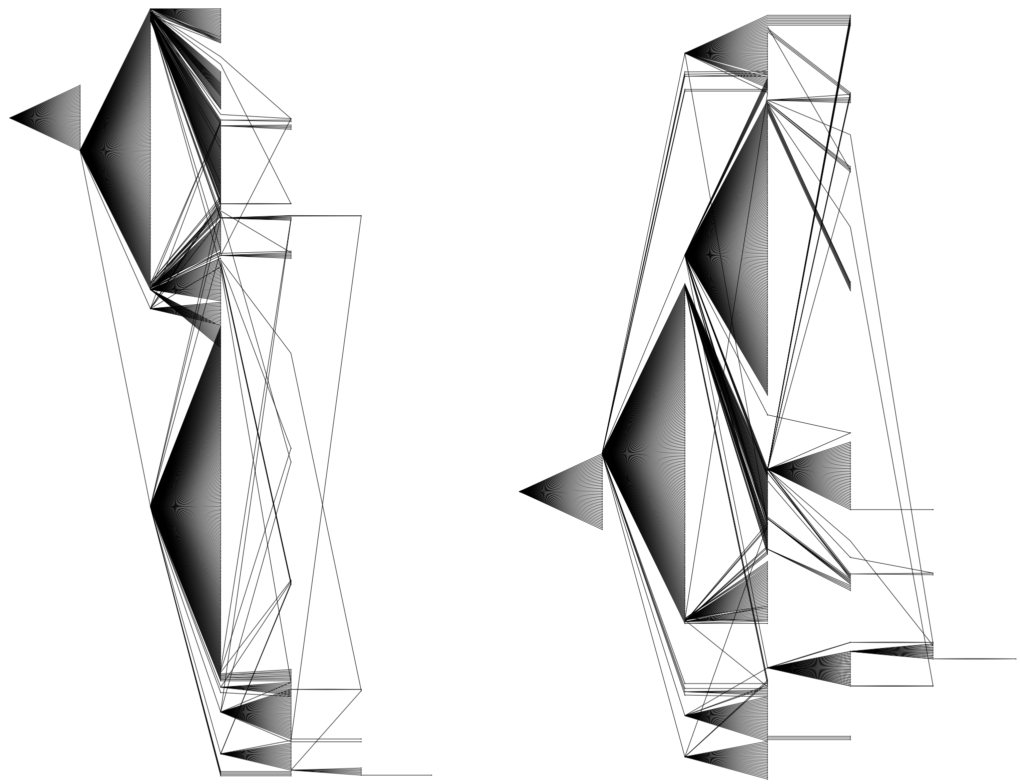


a. SML sub-library b. CMU sub-library c. Edinburgh sub-library

Figure 5.4: Graphs of indexes for the three sub-libraries using the *EQ* index pair.

Figure 5.4 shows the resulting indexes for each of the three sub-libraries using the *EQ* index pair. These graphs illustrate the general shape of the indexes for each of the sub-libraries and point out some differences among them. The SML sub-library (Figure 5.4a) has almost no hierarchy to it (only four nodes at the third level). The CMU sub-library (Figure 5.4b) is also shallow (three levels) but has a few more nodes with children and more nodes at the third level. In contrast, the Edinburgh sub-library (Figure 5.4c) has six levels and many more nodes with children; since many of the function types in the Edinburgh sub-library are polymorphic, they can be instantiated to another type in $match_{gen}$. For example, the bottom-most third level node in Figure 5.4c (the one with the most children) has type $\alpha \rightarrow \beta \rightarrow \gamma$. As illustrated in Figure 5.3, we could add depth to the flatter indexes by introducing special nodes.

The overall structure of the indexes for each sub-library and the combined Community Library do not change significantly between using *EQ* and *R-U* for the index pair. Figure 5.5



a. *EQ* b. *R-U*

Figure 5.5: Graphs of indexes for the Community Library.

shows the graphs of the indexes for both the *EQ* and *R-U* index pairs for the Community Library.

Table 5.4 summarizes the statistics for the indexed libraries generated by applying the *EQ* and *R-U* index pairs to each of the sub-libraries and to the Community Library, as we have described. Using equivalence classes roughly halves the number of nodes in an indexed library from the number of components in the original library. There is not much further compression gained from adding the reorder and uncurry relaxations. It is possible, however, that a more diverse library could gain additional compression by using the *R-U* index pair. Because of the limited size of the libraries in the table, we should not draw any "deep" conclusions from the statistics on these index structures; the information here is meant to show the results of building such structures.

Figure 5.6 shows the details of some of the nodes with more than a few children and some of the shared nodes in the index for the Community Library (using the *EQ* index pair). Other nodes are compressed into diamonds labeled with the number of other nodes not shown. The number in a diamond node that is the child of another diamond node is the number of nodes that are children of *a* node in the parent diamond (the child nodes are not necessarily children of *all* nodes in the parent diamond). `'a`, `'b`, and `'c` denote type variables. Let us consider

	SML		CMU		Edinburgh		Community	
	EQ	*R-U*	*EQ*	*R-U*	*EQ*	*R-U*	*EQ*	*R-U*
# of elements	385	385	431	431	923	923	1739	1739
# of nodes	223	219	236	229	411	384	807	757
Ave. # elements / node	1.7	1.8	1.8	1.9	2.2	2.4	2.1	2.3
depth of structure	2	2	2	2	5	5	6	6

Table 5.4: Statistics on indexes for the Community Library and sub-libraries.

an example of how to use this index both to improve the efficiency of retrieval and to browse the library. Suppose we want a function to concatenate two lists. We use the query Q = `'a list * 'a list -> 'a list` with exact match. Matching the query against nodes rather than components cuts the number of attempted matches roughly in half (807 matches instead of 1739). Pruning based on $M_>$ reduces this further. For example, since neither $match_{gen}$(Q, `'a -> 'b -> 'c`) nor $match_{gen}$(`'a -> 'b -> 'c`, Q), there cannot be a node that is a descendant of `'a -> 'b -> 'c` that matches with Q, so we prune all the descendants of the node `'a -> 'b -> 'c` (360 nodes). Using the same reasoning, we also prune descendants of all other nodes at the third level except `'a * 'b -> 'a` and `'a * 'b -> 'b` (another 169 nodes). Thus, between combining equivalent components and pruning subtrees of the indexed library, we reduced the number of attempted matches from 1739 to only 278.

The final result of the query is the node with signature `'a list * 'a list -> 'a list`, which contains five elements (functions with that signature): *interleave* and @ from list modules in the Edinburgh sub-library; and *union*, *intersection*, and *difference* from a set module in the CMU sub-library. Suppose we now wanted to browse other, related functions. Starting at the `'a list * 'a list -> 'a list` node in the graph, we could look at the next most general nodes (i.e., the parents of the node), namely `'a * 'b -> 'a` and `'a * 'b -> 'b`. We might then choose to look at the next most general node, or at other children of `'a * 'b -> 'a`. Other children of both `'a * 'b -> 'a` and `'a * 'b -> 'b` include nodes that contain functions to concatenate a variety of types, including strings, vectors, arrays, sets (union), and splay trees (join). Without the index, it would be much harder to find functions related to our retrieval result at all.

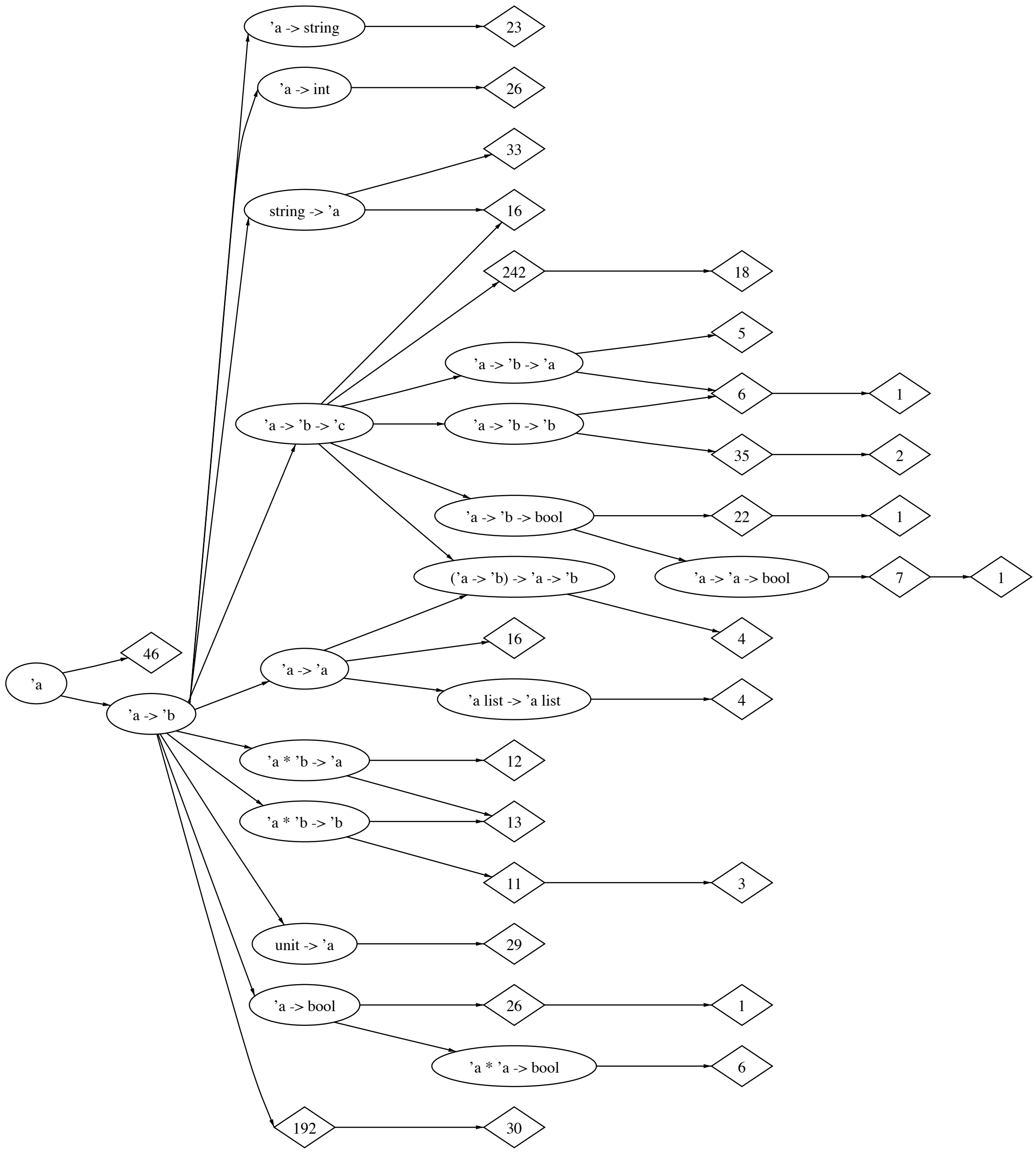


Figure 5.6: Details of some nodes in Community Library (*EQ* index pair).

5.2.3 Discussion

While it is not possible to assume that all libraries will have exactly the same characteristics as the Community Library, we expect that the main characteristics of the function types will be the same for most libraries. There will be some components with equivalent types, and hence compression of components into nodes. There is also not likely to be more depth, so we expect most indexes to be relatively shallow. In particular, libraries with no polymorphic types at all are completely flat. Even in shallow indexes, however, using the indexed library will reduce the number of matches required for retrieval as a result of the compression of components into nodes. Moreover, for queries where we can prune a large sub-branch, we reduce the number of matches dramatically.

To take advantage of indexed libraries for these improvements in storage and retrieval, the accesses or retrieval match must be the same as the equivalence or generality matches used to build the index. The choice of index pair depends on the intended use of the library. If we only expect to use exact match to retrieve from the library, for example, than we should use the EQ index pair.

If we cannot make any assumptions about which matches will be used to retrieve from the library, then we can use the most relaxed index pair (with relaxations type constructor, reorder, and uncurry).[1] In this case, we only need one index, but also have to match against each element in a matched node for retrievals using a stricter match. For any stricter match M that uses a subset of the relaxations that were used to build the library (i.e., any match that does not use the specialized relaxation), the elements that match a query under M will be a subset of the elements that match the query using all the relaxations, and hence all matches will be in the same node.

Indexing is probably not the best way to browse a library, since indexed libraries are likely to be shallow, with a few nodes that have a lot of children (e.g., nodes of type $\alpha \rightarrow \beta$ and $\alpha \rightarrow \beta \rightarrow \gamma$). Adding special nodes does not improve the structure very much. We would prefer a deeper structure with fewer children at each node. When browsing, a user needs to be able to choose a particular node from all the children of a node, a task that is much more difficult if a node has 100 children than if it has 10 children.

5.3 Substitution

Substitution applications answer questions of the form "Can we substitute component C for component Q?". Examples of substitution questions include

- If I replace Q with C in a piece of code, will the code still type check?
- If I replace Q with C in a piece of code, will the code still have the same observable behavior?
- If C and Q are specifications of object types, is C a subtype of Q?

We use the various signature and specification match definitions to answer these questions. Signature matching addresses the first question, specification matching the second. To answer

[1] We did not build indexes for the Community Library using this index pair because the type constructor relaxation was not part of our signature matcher at the time, so we used only the reorder and uncurry relaxations.

the third question, we use either signature or specification module matching to model subtyping, depending on which definition of subtyping we use. For a match $M(C, Q)$, C is the component we would like to substitute for Q; we treat Q as the "standard" we expect C to meet.

5.3.1 Substitution Guarantees

The match definitions give us a range of guarantees about what conditions hold if we substitute a component C for another component Q.

Signature matches generally verify that we can interchange components without type errors (potentially modulo some transformations). In particular, consider the case where C and Q are functions. Obviously, if C's signature matches Q's signature exactly, we can "plug" C in directly (either by calling the function or by cutting and pasting the actual code), and the code is guaranteed to type check. In ML, the same holds for generalized match. For reorder, uncurry, or type constructor match, we would need to change the order of tuple arguments, the form of the arguments, or the name of the user-defined types, respectively. The signature matcher actually calculates this information in determining the match, and thus could automatically generate "wrapper" functions that would convert from the form expected by Q to the form expected by C. The only relaxed match for which we cannot easily guarantee type correctness is specialized match. If C matches with Q under specialized match, Q is more general than C. Plugging in C for Q directly instantiates some of the type variables of Q, which may break the type correctness of something else that relies on the more general type of Q.

If C and Q are modules, module signature match with function match $match_{fn}$ guarantees type checking modulo function names (and modulo whatever transformations are necessary to ensure function type checking for $match_{fn}$).

Specification matches provide a range of guarantees about a program's behavior when substituting C for Q. In particular:

- If exact pre/post match holds on C and Q, then C and Q are behaviorally equivalent under all conditions; using C for Q should be transparent.
- If exact predicate or plug-in match holds, then C can be substituted for Q and the behavior specified by Q will still hold, although we are not guaranteed the same behavior when Q_{pre} is false.
- If weak post match holds, then the specified behavior holds when S_{pre} is satisfied. Depending on the context, we may be able to ensure that S_{pre} holds and hence guarantee the behavior specified by Q.

Specification matching is thus a cheap approximate method of program verification: Suppose we have a component S that we want to use to implement something specified by Q. We use exact pre/post match to verify that S satisfies Q. If exact pre/post match is too strong, we can use one of the other matches for a weaker but still useful guarantee. Of course, we are assuming that the implementation of S satisfies its specification.

5.3.2 Subtyping

A particular case of substitution is subtyping. In object-oriented programming languages, an *object type*[2] defines a collection of *objects*, which consist of *data* (state) and *methods* that act on the data [Car89, Ame91, Mey88]. Intuitively, a type σ is a subtype of another type τ if an object of type σ can be substituted for an object of type τ.

Precise definitions of subtyping vary in the strictness of this notion of substitutability from simply requiring the methods' signatures to match (*signature subtyping*) to requiring a correspondence between formal specifications of methods (*behavioral subtyping*). In the remainder of this section, we relate these definitions of subtyping to signature and specification matching.

To use signature and specification matching to model signature and behavioral subtyping, we must convert object types to our context. We base our definition of an object type specification on that of Liskov and Wing [LW94].[3] An object type specification includes the following information:

- The object type's name
- A description of the object type's value space
- For each of the object type's methods m_i
  - Its name
  - $m_i.sig$ – its signature
  - $m_i.spec$ – its behavior in terms of pre- and post-conditions

We model this as a module specification with a type declaration for the object type, a global variable of the object type to hold the current state of the object (an element of the value space), and a function specification for each method.

For example, Figure 5.7 shows the module specifications for two objects. The first is *BagObj*, a mutable bag object with global variable b and methods *put*, *get*, and *card*. The clause **modifies** b in the functions *put* and *get* indicates that the value of b may be changed by the functions. In the **ensures** clauses, we use $b\%$ to stand for the value of the bag in the final state and b for the value in the initial state. The second specification is of a stack object. *StackObj* is based on the same trait as bag, but has a stricter specification for the method that removes an object (*pop_top*) and an additional method, *swap_top*. In keeping with the Liskov and Wing approach,

[2] These are usually simply called "types", but we need to distinguish types of objects from types in signatures.

[3] We differ from Liskov and Wing in that we do not include invariants or constraints. We focus here on modeling certain aspects of object specifications in our framework.

```
signature BagObj = sig
    (*+ using Container2 +*)
    type α t (*+ based on
      Container2.E Container2.C +*)

    val b : α t

    val put : α → unit
    (*+ put (e)
      modifies b
      ensures b% = insert(e, b) +*)

    val get : unit → α
    (*+ get ( ) = e
      requires not(isEmpty(b))
      modifies b
      ensures (b% = delete(e, b)) and
              (isIn(e, b)) +*)

    val card : unit → int
    (*+ card ( ) = n
      ensures n = size(b) +*)
  end
```

```
signature StackObj = sig
    (*+ using Container2 +*)
    type α t (*+ based on
      Container2.E Container2.C +*)

    val s : α t

    val push : α → unit
    (*+ push (e)
      modifies s
      ensures s% = insert(e, s) +*)

    val pop_top : unit → α
    (*+ pop_top ( ) = e
      requires not(isEmpty(s))
      modifies s
      ensures (s% = butLast(s)) and
              (e = last(s)) +*)

    val swap_top : α → unit
    (*+ swap_top (e)
      requires not(isEmpty(s))
      modifies s
      ensures s% = insert(e, butLast(s)) +*)

    val height : unit → int
    (*+ height ( ) = i
      ensures i = size(s) +*)
  end
```

Figure 5.7: Larch/ML specifications of bag and stack object types

we assume that create methods are defined elsewhere. Appendix B lists the *Container2* trait on which both specifications are based.

The *StackObj* specification differs in several ways from the *Stack* specification in Figure 3.1 (pg. 40). First, in *StackObj*, stacks are mutable, whereas in *Stack* they are not. Because the *Stack* specification in Chapter 3 specifies the behavior of a typical implementation in a functional language, its stacks are immutable. Here, however, we wish to model the specification of a stack in the object-oriented paradigm, and hence these stacks are mutable. Second, *Stack* has separate functions for *pop* and *top* while *StackObj* combines these in *pop_top*. Again, this is mainly a by-product of the difference between a functional implementation and an object-oriented one.

Third, each specification has additional functions that the other does not.

We now consider how to define the subtype relation between two objects (modules). We define three different subtype relations: one based on signature matching and two based on specification matching. Let T represent the module interface of the supertype and S the module interface of the subtype. All subtype definitions require a correspondence between each method in T and a method in S but allow additional methods in S. The correspondence between methods varies among the subtype definitions but is always a function match definition. Thus, all three subtype definitions have the following general form:

Definition 5.3.1 (Generic Subtype)
$Subtype(S, T) = M\text{-}match_{gen}(S,\ T,\ match_{method})$

S is a subtype of T if S matches with T under generalized module match. The particular notion of subtyping depends on $match_{method}$, the match used at the method (function) level. We instantiate $match_{method}$ with the appropriate function match definition for each of the three different subtype definitions.

Signature Subtyping

Most definitions of subtyping use the contravariant and covariant rules for function (method) signature match [Car89, Ame91]. For each method, m_τ, in the supertype, there is a method m_σ in the subtype such that m_τ's input types are subtypes of m_σ's input types (contravariance) and m_σ's return type is a subtype of m_τ's return type (covariance). We assume that there is a subtyping relation $<_{ST}$, which includes an assumed subtype relation between the types currently under consideration.

We define contra/covariance subtyping of object types by instantiating $match_{method}$ in the generic subtype definition (Definition 5.3.1) with contra/covariance function match. The contra/covariance function match is similar to the specialized function signature match except that we must reverse the ordering relation ($<_{ST}$) for the return type.

Definition 5.3.2 (Contra/Covariance Signature Subtype)
$Subtype_{c/c}(S, T) = M\text{-}match_{gen}(S_{sig},\ T_{sig},\ match_{c/c})$

where $match_{c/c}(\sigma, \tau) =$
for $\sigma = (\sigma_1 \ldots \sigma_{n-1}) \rightarrow \sigma_n$
and $\tau = (\tau_1 \ldots \tau_{n-1}) \rightarrow \tau_n$
$\tau_i <_{ST} \sigma_i \ \ \forall\ 1 \leq i \leq n-1$ and
$\sigma_n <_{ST} \tau_n$

Under contra/covariance signature subtype, *StackObj* is a subtype of *BagObj* using the obvious mapping of bag functions to stack functions, since for all three functions, the types match exactly. Note that if we remove the *swap_top* method from *StackObj*, *BagObj* is also a subtype of *StackObj*, even though using the *get* method from *BagObj* does not guarantee stack-like behavior. This illustrates the need for a stronger notion of subtyping based on the behavior of the methods.

Specification (Behavioral) Subtyping

More recent work on subtyping has focused on adding semantic information to more precisely capture the notion of substitutability of a subtype, as defined by Liskov in her OOPSLA '87 keynote address [Lis87]:

> If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.

Behavioral notions of subtyping that attempt to capture this substitutability property have since been defined by many [Ame91, DL92, Lea89, LW90, LW94, Mey88]. There are subtle differences between all these subtype definitions, but common to all is the use of pre-/post-condition specifications both to describe the behavior of types and to determine whether one type is a subtype of another. Let m_T be a method of supertype T, and m_S be the corresponding method of subtype S. Then America [Ame91], for example, defines subtype in terms of the following pre-/post-condition rules[4] for each method of the supertype:

- *Pre-condition rule.* $m_T.pre \Rightarrow m_S.pre$
- *Post-condition rule.* $m_S.post \Rightarrow m_T.post$

which is exactly our plug-in match. As with signature subtyping, behavioral subtyping requires that each method in the supertype T have a corresponding method in the subtype S, but there may be additional methods in S. We define behavioral subtyping by instantiating $match_{method}$ in the generic subtype definition (Definition 5.3.1) with plug-in function specification match (Definition 3.2.4, pg. 44). We assume that the signatures match.

Definition 5.3.3 (Behavioral Subtype)
$Subtype_{behav}(S,T) = M\text{-}match_{gen}(S_{spec},\ T_{spec},\ match_{plug-in})$

Another, slightly weaker pair of pre-/post-condition rules allows an additional assumption about the pre-condition in the post-condition rule:

- *Pre-condition rule.* $m_T.pre \Rightarrow m_S.pre$
- *Post-condition rule.* $(m_S.pre \wedge m_S.post) \Rightarrow m_T.post$

[4] We omit the abstraction function for simplicity.

This post-condition rule is the same as our weak post function specification match, and is used for the same reason: to allow us to prove the post-condition relation when it is necessary to make an assumption about the pre-condition. We define a new subtype definition that uses the pre-/post-condition rules above:

Definition 5.3.4 (Weak Behavioral Subtype)

$Subtype_{weak-behav}(S, T) = M\text{-}match_{gen}(S_{spec},\ T_{spec},\ match_{weak\text{-}plug\text{-}in})$

where

$$match_{weak\text{-}plug\text{-}in}(S, Q) = (Q_{pre} \Rightarrow S_{pre}) \wedge ((S_{pre} \wedge S_{post}) \Rightarrow Q_{post})$$

Consider the *StackObj* and *BagObj* specifications in Figure 5.7. We would like to show that *StackObj* is a behavioral subtype of *BagObj*. As the objects are specified, we cannot show the stronger behavioral subtype relation (Definition 5.3.3), because we cannot prove $match_{plug-in}(pop\_top, get)$, since we cannot reason about the case where the stack or bag is empty. However, we can show that *StackObj* is a weak behavioral subtype of *BagObj* (Definition 5.3.4), since the weak plug-in definition specifically allows us to exclude the case where the stack or bag is empty.

To show $Subtype_{weak-behav}(StackObj, BagObj)$ (or equivalently, $M\text{-}match_{gen}(StackObj_{spec},$ $BagObj_{spec},\ match_{weak\text{-}plug\text{-}in}))$, we must define the mappings U_F and U_{TC} to satisfy the three requirements of module match.

There is only one user-defined type in both *StackObj* and *BagObj*, and it is the same (i.e., $UserOp(\Sigma_{BagT}) = UserOp(\Sigma_{StackT}) = t$). So U_{TC} is the identity function ($U_{TC}(t) = t$). We define U_F as follows: $U_F(put) = push$, $U_F(get) = pop\_top$, and $U_F(card) = height$. U_{TC} and U_F satisfy the three requirements of generalized module match:

(1) U_{TC} and U_F are both one-to-one total functions. (U_F is not onto, but does not need to be for generalized module match.)

(2) $match_E(\alpha\ t, \alpha\ t)$

(3) $match_{weak\text{-}plug\text{-}in}(push, put)$
$match_{weak\text{-}plug\text{-}in}(pop\_top, get)$
$match_{weak\text{-}plug\text{-}in}(height, card)$

We translated our specifications of *StackObj* and *BagObj* into LP input and were able to prove the weak plug-in matches with very little user guidance. Figure 5.8 shows the LP proof script to load the specifications and prove the weak plug-in match between each pair of methods. The proofs for $match_{weak\text{-}plug\text{-}in}(push, put)$ and $match_{weak\text{-}plug\text{-}in}(height, card)$ are trivial, since the specifications are identical modulo variable names. The proof for $match_{weak\text{-}plug\text{-}in}(pop\_top, get)$ requires an additional lemma and some guidance. Appendix B shows the *Container2* trait on which both *BagObj* and *StackObj* are based, as well as `bagobj.lp` and `stackobj.lp`, the result of translating *BagObj* and *StackObj* into LP input.

```
thaw Container2_Axioms

%% execute bagobj.lp
%% execute stackobj.lp

% weak-plug-in(push, put)
prove (putPre => pushPre) /\ ((pushPre /\ pushPost(b, b', e)) => putPost(b, b', e))
  [ ] conjecture

% weak-plug-in(height, card)
prove (cardPre => heightPre) /\ ((heightPre /\ heightPost(b, i)) => cardPost(b, i))
  [ ] conjecture

% Additional lemma assert 0 <= count(e,s)
prove delete(e,insert(e,s)) = s
  apply Container2.2 to conjecture
  [ ] conjecture

% weak-plug-in(pop, get)
prove
  (getPre(b, e) => popPre(b, e)) /\
  ((popPre(b,e) /\ popPost(b, b', e)) => getPost(b, b',e))
  ..
  resume by induction on b
    <> basis subgoal
    [ ] basis subgoal
    <> induction subgoal
    [ ] induction subgoal
  [ ] conjecture
qed
```

Figure 5.8: LP subtype proof script

5.3.3 Discussion

Section 5.3 shows how to use our match definitions, particularly specification match, to show that one component may be substituted for another. Here are two scenarios that illustrate how we can use substitution in practice.

Scenario 1. Suppose that as part of a system implementation we need a component that we have specified with a module specification Σ_Q. Further, suppose that there is a module in our library with specification Σ_L, and that the implementation of Σ_L has been verified to be correct with respect to the specification Σ_L. If we can show that Σ_Q is matched by Σ_L under generalized module match with plug-in function match, then we know that we can use the library module and the behavior will be consistent with that specified by Σ_Q. Thus, we use specification match to check that using a library component will not "break" our system.

Scenario 2. Suppose that we have a piece of software that includes a component specified by Σ_Q and that as part of the maintenance of the software, we need to replace the component with an upgrade specified by Σ_L. If we can show that Σ_Q is matched by Σ_L under generalized module match with plug-in function match, then we know that replacing the component with the upgrade will not change the observable behavior of the software. Thus, we use specification match to check that upgrading will not "break" our system. An advantage of this scenario is that the specification Σ_Q is known to the developer of the upgrade, and thus it is reasonable to assume that both specifications are based on the same trait (hence base terms are the same). Further, the specifications should be very similar (assuming the functionality of the component did not change), and thus the match should be easy to prove.

Section 5.3.2 shows how the method rules of signature and behavioral subtyping are instances of our more general notion of signature and specification matching. Our definitions of behavioral subtyping do not capture the ideas of other work in the area completely, however. First, we do not address explicitly the use of an abstraction function for cases when the value spaces of the subtype and supertype differ. We could, however, include an abstraction function in the specification of the subtype and explicitly map the values. Alternatively, we could use relaxed signature matches in some cases (for example, when the value space of the supertype is more general than the value space of the subtype). Second, we do not handle invariants or constraints in our specifications, although it should be possible to add this in our framework by extending Σ_T to include constraint specifications in addition to user-defined type declarations. A third and more important exclusion in our definitions is the lack of a way to model additional methods in the subtype in terms of methods in the supertype (the extension rule in Liskov and Wing[LW94]).

What we have shown is how subtyping fits into our framework of signature and specification matching. Subtyping based on just signatures is subsumed by module match using function signature matching, and with minor or no variation, we define the core part of many behavioral notions of subtyping with module match using function specification matching. Additionally, by using this framework, we provide tools to automate subtype checking.

Chapter 6

Related Work

Related work generally divides by the application. There is no other work that applies a uniform approach for all the applications we describe in the thesis, although a few consider both retrieval for reuse and indexing. The primary area of related work addresses retrieval for reuse. We divide work on retrieval for reuse further into signature-based retrieval (Section 6.1), specification-based retrieval (Section 6.2), and other retrieval approaches (Section 6.3).

Our work on signature and specification matching for retrieval is unique in three ways. First, for functions, we have identified a small set of matches, each of which identify an intuitive correspondence between components that are similar but not identical. The matches are presented in a general framework that allows composition of function signature matches and allows us to describe much of the related work within our framework. Other work chooses one (or sometimes two) matches, and is not easily extensible to other matches as ours is. We talk more about the advantages of our match definitions separately for signatures and specifications.

Second, most related work has focused on matching at the function level (with the exception of [CHJ93, SC94] for signatures and [JC95] for specifications). We extend matching to the module as well. Moreover, since we define all our function match definitions to follow a common form, we are able to use function match as a parameter to module match, and hence define both signature and specification module match with one parameterized definition. Thus, our definition of module match is more flexible than the limited treatment in the few systems that consider modules at all.

Finally, we go beyond retrieval for reuse and present three more retrieval applications (statistical analysis, browsing, and compound retrieval), and two other applications (indexing and substitution). Other work focuses primarily on retrieval for reuse, although a few have a notion of indexing [RT89, JC95, MMM94].

6.1 Signature-Based Retrieval

Signature matching for retrieval was first proposed concurrently by Rittri [Rit89] and by Runciman and Toyn [RT89]. Most related work on signature matching has focused either on signature matching as an application of a particular theoretical definition of type isomorphism or as a very basic retrieval tool to be used in conjunction with other tools.

Our work on signature matching (initial results published in [ZW93] with an extended version showing the new applications in [ZW95a]) is unique because (1) it takes a "pick and choose" approach to function match definitions, (2) it includes module matching, and (3) it includes applications other than retrieval for reuse. The second and third points are discussed for both signatures and specifications at the beginning of this chapter. We elaborate here on the first point. We have identified a small set of primitive function matches based on transformations of types. The other work identifies and implements a single function signature match. With our approach, we can describe related work on signature matching in terms of our framework and definitions (as elaborated below). We require one new match definition for unification, which allows variables to be instantiated in both types with the same substitutions:

$$match_{unify}(\tau_l, \tau_q) = \exists \text{ a sequence of variable substitutions, } U, \text{ such that } match_E(U\ \tau_l, U\ \tau_q)$$

Our approach also supports orthogonality of concepts, allowing the user to pick and choose whichever match is desired, perhaps through a combination of more primitive matches. Each transformation corresponds to an intuitive relationship between two types (e.g., reordering of elements in a tuple). The only other work to take the same approach as ours is that of Stringer-Calvert [SC94], whose work was inspired by ours. His match definitions are based on our definitions of exact, reorder, generalized, and specialized match (he does not include uncurry and type constructor matches); he defines an additional *subset* match, which is like reorder match but allows tuple elements to be dropped. He has implemented a signature matcher for Ada types.

An additional distinction of our signature matching approach is that we chose our framework and definitions with an eye toward how the matches would be used in practice. In addition to examples that illustrate the match definitions, we provide a collection of experiences from actual use of the system on a moderately-sized library. Although other systems have been implemented, at least as prototypes, there are few other examples of real use.

6.1.1 Category Theoretic Approaches

Research using category theory takes the approach of identifying a category and a set of axioms that are sound and complete for the category (i.e., the axioms find an equivalence between two types if and only if the types are isomorphic in the category), and using the isomorphisms defined by the category as the basis for function type match and retrieval. Results vary based on the category and axioms used, and based on whether variables can be instantiated in the query (matching), in both the query and the library component (unification), or not at all (equality). This approach leverages off the extensive work in the decidability and complexity of unification and matching of isomorphisms in various categories, but the extra axioms required for completeness give rise to some isomorphisms that are surprising in the context of type matching (e.g., $\alpha \rightarrow unit$ is isomorphic to $unit$).

Rittri (along with Runciman and Toyn) was the first to propose using function signatures for retrieval. In his first work [Rit89, Rit91], he implemented a system to retrieve types that are isomorphic to the query type in Cartesian closed categories. This isomorphism allows equivalence between types that differ only in their currying or argument order. The match is similar to $match_{reorder^+} \circ match_{uncurry^+}$, but also admits some equivalences to get soundness and completeness that do not make much intuitive sense in the context of types. For example, in Rittri's system, $unit * \alpha$ is isomorphic to α, and $\alpha \rightarrow unit$ is isomorphic to $unit$. The implemented retrieval system was for a restricted type system (only type variables, $unit$, function application, and tuples).

Rittri then extended the system to also retrieve more general types from the library modulo the isomorphism [Rit90, Rit92b] (i.e., similar to our $match_{reorder^+} \circ match_{uncurry^+} \circ match_{gen}$, and extended the type system to allow user-defined type constructors.

Rittri's third system [Rit92a] makes two changes from the previous system. First, he restricts isomorphisms to *linear isomorphisms*, which eliminate some of the un-intuitive equivalences (e.g., $\alpha \rightarrow unit$ is not linearly-isomorphic to $unit$). Second, he retrieves types that are *unifiable* with the query modulo linear isomorphism (rather than just those that are more general). By allowing unification, he is able to match types with differing numbers of tuple elements, since a tuple element can be instantiated with $unit$ and reduced away. For example, $\alpha * int \cong unit * int \cong int$ (by substituting $unit$ for α and then applying the axiom for tuples that contain unit).

Di Cosmo extends Rittri's approach with a theory that also handles isomorphisms of types with *let* expressions[DC92]. His implementation of retrieval is the only one we know of that is widely available; it is distributed with the CAML-Light system [Ler95a]. There are two different searches available – equality modulo isomorphism (no instantiation of variables) and matching modulo isomorphism (which retrieves more general types).

6.1.2 In Conjunction with Specification Match

Some specification-based retrieval work has an explicit notion of signature match [RW91, SGS91]. We discuss the details of the signature matches for these systems along with their specification matches in Section 6.2.

Chen, Hennicker, and Jarke [CHJ93] describe a framework for both signature and specification matching, but have only implemented signature matching so far. Components are specified in the algebraic specification language ASL; modules consist of a set of sorts, a set of operations on the sorts, and a set of axioms about the operations. They define an *implements* relation: S is an implementation of Q if the signatures match and $Mod(S) \subseteq Mod(Q)$. $Mod(S)$ is the class of models in which the axioms of S are satisfied. Two components S and Q match if their signatures match and if $Mod(S) \subseteq Mod(Q)$, but currently only the signature match part is implemented. Their signature match definition is the same as generalized module match with function match $match_{reorder}$. Their implementation is interesting: library components are stored in a database, and a query signature is translated as into a database query to the knowledge base management system.

6.1.3 Others

Runciman and Toyn [RT89, RT91] approach retrieval from a slightly different angle. They assume that queries are constructed by example or by inference from context of use and match a query and component if they are unifiable (i.e., $match_{unify}$). There can be multiple queries for a particular retrieval. One focus of their work is to reduce the search space for retrieval. They use *initial/final indexes* as abstractions of the type. Indexes of types form a tree over the library. These indexes would not easily generalize to allow other matches like uncurrying.

Runciman and Toyn also define the notion that one type is an *applicative instance* of another,[1] and discuss using this (partial order) relation to explore a library (i.e., as an index), but they do not use it in conjunction with search.

6.2 Specification-Based Retrieval

Work on specification matching for retrieval varies widely in the kinds of specifications used, in the expressiveness of the match, and in the reasoning power of the implementation. We divide the work into approaches that use specifications with separate pre- and post-conditions, and those that do not, since explicit pre- and post-conditions allow matches like plug-in, which we feel are the most important.

As summarized in the previous section, our work (initially published in [ZW95b]) differs in three ways. We elaborate here on the difference in the function specification match definitions. Our general approach allows us to define and relate multiple matches within the lattice of matches. Our implementation lets us easily experiment with the different definitions. The matches correspond to intuitive ideas about how two components relate (indeed, plug-in match is used as a definition in several of the other systems). The intuitiveness of some of the other match definitions (e.g., [JC95, MMM94]) is not clear in some cases. We feel that this could hamper the willingness of a user to reuse a retrieved component, if he or she cannot understand

[1] Intuitively, a type τ is an applicative instance of another type σ if τ could be defined as an application of σ (e.g., *map*: $(\alpha \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta\ list$ is an applicative instance of *map2* : $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha\ list \rightarrow \beta\ list \rightarrow \gamma\ list$).

how the component is related to the query.

An additional distinction between the various approaches is the "power" of the match engine. Most approaches [KYS85, KRT87, RW91, PP93] use a restricted form of reasoning about the match for pragmatic reasons. For example, Rollins and Wing use higher-order unification to implement the match definition, which allows them to leverage off λProlog, but cannot support equational reasoning in proving a match. Only recently [MMM94, FKS94, ZW95b] have researchers begun to push theorem proving technology to do more powerful reasoning for specification-based retrieval.

6.2.1 Pre/Post Style Specifications

Rollins and Wing

The inspiration for our work comes from Rollins and Wing [RW91], who first proposed the idea of function specification matching. They implemented a prototype system using λProlog as the specification and query languages. The system provides both signature and specification matching for ML functions, using λProlog's higher order unification for the matching.

The signature match is the same as our function signature match with relaxations uncurry, reorder, and unify. They handle a subset of the ML type system that includes type variables but not user-defined types.

Specifications are written in a Larch style, with a shared component and an interface component. A function component matches a query component if their signatures match and their specifications match. A function specification matches a query specification if the query pre-condition implies the function pre-condition, and the function post-condition implies the query post-condition (i.e., plug-in match). One limitation of this approach is that λProlog does not use equational reasoning, and so the search may miss some functions that match a query but require the use of equational reasoning to determine that they match. For example, if a query that specifies that a function returns an empty container (s) with the clause *isEmpty(s)* and a library function specifies the same thing with $length(s) = 0$, λProlog cannot determine that they match.

Rollins and Wing also have a rule to match a query with the composition of two library functions. For example, the signature query $int\ list \rightarrow int\ list$ is matched by applying *gensort*: $(\alpha * \alpha \rightarrow bool) \rightarrow (\alpha\ list \rightarrow \alpha\ list)$ to *lessthan*: $int * int \rightarrow bool$.

Inquire

Perry's Inscape system [Per89] is a specification-based software development environment. Its Inquire tool [PP93] provides predicate-based retrieval in Inscape. Components are either operations, data objects, or modules. Operations are specified with pre-conditions and post-conditions in first order logic. (There are two kinds of post-conditions: predicates that are true as a result of executing the operation, and predicates that must eventually be satisfied *obligations*.) For operations, match is either exact pre/post or a form of generalized match. Inquire also allows specifications of data objects, and thus can retrieve data objects as well. The prototype system has a simplified and hence fairly limited inference mechanism. In Inscape, the user must provide specifications for each component anyway, so the query for a retrieval will already be written. If no existing library components match, the user will start from the specification to implement the component.

VCR

A recent project that has had encouraging practical results is the VDM-based Component Retrieval System (VCR) [FKS94], which is part of the NORA software development environment [SGS91]. This system is closest to ours in approach. Components are Modula-2 functions that are specified in VDM. The library is the Lins Modula-2 library (50 modules, 1000 procedures), of which they have specified about half. Match is a multi-step process. The first step is signature matching. They use $match_{reorder} \circ match_{unify}$ as the signature match (they allow variables in the query types to allow incomplete specifications).

Specification matching uses the $match_{plug\text{-}in}$ definition. A focus of this work is on efficiency of proving match; the tool performs a series of filtering steps before doing the specification match. In addition to the signature match, there is also a model checking step, which eliminates obvious non-matches. The model checker tests the match obligations in finite models – for example, they report good results from modeling integers as either 0 or 1, and lists as either nul or a list of one element. For the actual proof of plug-in match, they use the OTTER theorem prover. They are very selective about the axioms provided with each proof obligation in order to keep the search space tractable.

Jeng and Cheng

Jeng and Cheng [JC92, JC95] define two different matches where components are specified using order-sorted predicate logic (OSPL). Components are modules that consist of inherit clauses and a set of function specifications. A function specification consists of a pre-/post-condition pair where terms are in OSPL. Both matches are instances of our generalized module match. In the first case (*relaxed exact match*), the function match is primarily syntactic. The user supplies a renaming of predicates, terms can be reordered in some cases (but not all), and a conjunction is matched if any of its subterms are matched.

The second match (*logical match*) is based on the subsumption relation between clauses. But the example indicates that most of the increased flexibility in the match (compared with relaxed exact match) comes from the type hierarchy (e.g., $int \leq real$) and from user-supplied relationships between terms (e.g., $size =_{pred} card$). Neither of these matches seem to correspond to any of the intuitive ideas about when two specifications should match.

Logical match is a partial order. Jeng and Cheng also describe how to use the match to build a hierarchy of library components. They have a prototype system, but do not describe their library or how they use the hierarchy.

6.2.2 Other Systems

Mili, Mili, and Mittermeir

Mili, Mili and Mittermeir [MMM94] define a specification as a binary relation that contains all the pairs of input and output that are correct for a function. A specification S *refines* another specification Q if S has information about more inputs and assigns fewer images to each argument. This is like plug-in match except that the match is in terms of relations rather than predicates. They use the refines relation to build a lattice of the library components and as the primary match definition. They define a relaxed match for the case where there is not a library specification that refines the query. Relaxed match returns the functions that satisfy "the largest portion of the requirements of the search key," by computing the meet of library specifications with the query specification, and finding the ones among those that refine all the other meets.

They describe an implementation of the match using the Otter theorem prover, and show results of building a lattice and doing a query over a library of twelve specifications of different Pascal compilers. We find that because the match definitions and lattice are all in terms of binary relations, it can be hard to get an intuitive feel for what the match means in terms of what the functions actually do.

Preliminary Approaches

Two earlier works have the same flavor as specification-based retrieval, but come at it from slightly different angles.

Katoh, Yoshida, and Sugimoto [KYS85] propose using English-like specifications and queries that are translated into first-order predicate logic formulas. They use "ordered linear resolution" to determine matching between a query and specification, and include relaxations for changing the order of parameters, making some parameters constants, or renaming subroutines. However, the match does not verify that the subroutines match and checks only for equivalence, not permitting any inference, and is hence closer in expressiveness to our signature matches.

The PARIS system [KRT87] maintains a library of partially interpreted *schemas*. Each schema includes a specification of assertions about the input and results of the schema and about how the abstract parts of the schema can be instantiated. Matching corresponds to determining whether a partial library schema could be instantiated to satisfy a query. The system does some reasoning about the schemas but with a limited logic.

6.3 Other Approaches

We view signature and specification matching as complementary approaches to more traditional information retrieval techniques. A user chooses the most appropriate tool for a task based on what information he or she has available and which tool is expected to give the best results for that particular task. A user can also use one tool as a filter for another. We categorize additional less closely related work based on the kind of information being matched (i.e., the kind of abstract).

Text-based Retrieval

The most common approach to software retrieval is text-based. Research in this area has applied techniques from information retrieval and relational databases to software retrieval. Queries and information on components are typically in a restricted keyword or attribute-value approach (often called *facets*). Matching corresponds to locating components in the system with the same or similar keyword/value pairs [AS87, FN87, PD89, MF93, SSS93, CE93, Pou93, MFCS95]. Another information retrieval approach extracts attributes from natural language documentation associated with each component [MBK91].

In some cases, additional structural information can be added in AI-based semantic net classifications. Information is either extracted from components and their documentation or generated by domain experts [HM91, FHR91, OHPDB92, FF93, Hen95].

The advantage to these approaches is that many efficient tools are available to do the search and match in these structures. A well-structured faceted classification also forms an index that can be used to browse the library. The disadvantage is that the characterization of the component's behavior is completely informal.

All of these other approaches require at the very least that the user learn the keyword language (except for the natural language-based approach in [MBK91]). Except for [MBK91] and [PP94], the information about the library components must be created by hand as well. Our work on signature matching uses a query language with which software engineers are already familiar – the programming language's type system.

Code-based Retrieval

Another class of matches [PP94, CMR92] allow queries over a representation of the component's actual code, e.g., abstract syntax trees. Such queries are useful for determining mainly structural characteristics of a component, e.g., nested loops or circular dependencies, but provide no support for browsing or indexing.

Example-based Retrieval

Another interesting approach to retrieval is a *query-by-example* technique [PP92, Hal93]. The user forms a query by giving examples of correct output for an input (or set of inputs). Matching involves executing library components to find one that generates the same outputs for those inputs (usually involving a basic notion of signature matching as a filter). In a sense, this is a form of specification matching, since the set of inputs and outputs specify the behavior of the component on at least some inputs.

Protocol-based Match (Interoperability)

There is a growing body of work about how to connect modules and determining whether two modules are interoperable. The notion of match is slightly different: looking at whether two components can be connected (i.e., plugged together) rather than whether one can be substituted for the other (i.e., plugged in). But the basic notions of a signature or specification to describe the module are the same.

Wileden et. al. survey *specification-level interoperability* [WWRT91]. Most work thus far has focused on signature-based interoperability, and how to convert types in a heterogeneous environment. Of particular interest is the ability to automatically generate the adapters that will interface between two components [Kon93, YS94, Tha94]. An additional concern when connecting modules is how the modules communicate or interact with one another. Allen and Garlan [AG94] use a subset of CSP to specify protocols for modules (ports) and the connections between modules (roles). A port matches a role (port-role compatibility) if the behaviors of the role include those of the port (within the context of the port's connection to the role). In our work, we assume that all communication is achieved with procedure calls, so we do not have a notion of protocol match.

The focus of each of these approaches is slightly different. Information retrieval approaches (with or without semantic nets) and query-by-example retrieval typically only address reuse as an application; code-based approaches focus mostly on browsing or analyzing the actual code; protocol-based approaches are concerned with interoperability, which is comparable to substitution. Of these approaches, only protocol matching could address all the applications covered by signature and specification matching. Text-based approaches lack the ability to express relations between the semantic properties of two components in order to do substitution. Code-based retrieval is not easily extensible to indexing or browsing. Example-based retrieval does not yield a relation between components other than exact match, so there is no relaxed retrieval and no way to generate an index. Protocol matching has a different focus, since two components are connected rather than substituting one for the other. But protocols are essentially an extended form of specification and hence fit within our general framework.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This dissertation lays the foundation for using semantic information to match software components and for using semantic matches for a variety of applications, particularly to utilize libraries of components more effectively. We present precise definitions of a variety of matches for signatures and specifications of functions and modules; we have implemented the various matches to use as a testbed for our ideas; and we show how the matches are used for a variety of applications.

Function signature matching provides a way to retrieve components. Function specification match addresses the problem of knowing when we can substitute one component for another. Module matching is good for cases where we are concerned with details of the interface of a module. The limitations of our approach to module matching indicate a need for a more abstract type system for modules.

Conclusion 1: Function signature matching should be a part of every software development environment and every software library interface.

Signature-based retrieval is an efficient, easy to use tool for locating functions of interest in a library. Using an indexed library makes retrieval even more efficient. Signature-based retrieval provides a way to locate functions for reuse in terms of the programmer's existing conception of the function, since programmers often think of a function in terms of what is input to the function and what the function returns. Further, signature-based retrieval is the only way to find functions whose types are more general than or are instances of another type. This is what makes browsing and statistical analysis possible.

An important factor in signature-based retrieval is controlling the number of functions retrieved by a query. If there are too many, a user will have difficulty choosing the relevant one (or ones). If there are none or no relevant ones, the user must try again. The key to controlling the number of matches is the appropriate use of relaxations. We can either allow the user to pick and choose relaxations (as we do in Beagle) and thus control the number of matches himself or herself, or we can use metrics about the ideal number of matches and automatically add or drop relaxations to get within a desired range of number of matches.

Conclusion 2: If cost is not a factor, specification matching is useful for determining when one component can be substituted for another.

The case for specification matching is less conclusive. With our approach, the cost of proving a match is too high to use specification matching for retrieval. Given a particular pair of functions or modules, however, specification match can prove that the behavior of a program will not change if we substitute one component for another, a property that can be used when deciding whether to use a library component, when determining subtype relations, or when upgrading an existing system with a new version of a component. Thus, the benefits of doing specification match indicate that the general approach has merit. However, to become commonly used, we must reduce the cost of doing the match. We discuss some ways of doing that in Section 7.2.2.

Conclusion 3: Match frameworks provide a general, highly extensible, and modular approach to the problem of matching.

Each kind of matching (function signature, function specification, and module) has a generic form that we instantiate for each particular match definition. These frameworks make it possible for use to compose function signature matches easily, relate the matches within each kind of match, define module match independent of the function match used, and extend the systems (e.g., with some of the things we discuss in Section 7.2).

Conclusion 4: Defining multiple matches provides necessary flexibility to address a range of applications.

Given a particular component granularity and kind of abstract (e.g., function signature), we define multiple matches rather than a single match. Initially, the reason for having multiple match definitions was to explore which relaxation (or combination of relaxations) is most useful. What we discovered, however, is that which match definition is the “right” one depends on the application and the context, and thus the right approach is to provide multiple matches and allow the user the freedom to specify which notion of match he or she wants.

Having multiple definitions is not uncommon. On a general level, signature and specification matching are multiple definitions of semantic matching, and semantic matching and text matching are multiple definitions of component matching. Even focusing on a particular description language, multiple notions of match abound. Consider the case of text matching. Notions of match include exact match, match using a thesaurus to identify similar terms, and regular expression match to find patterns. Even within regular expression matching, there is disagreement over what the "right" match is. The Unix commands `grep` and `egrep` have different regular expression languages, which is analogous to having different kinds of relaxed matches. Another indication that we need multiple match definitions comes from related work in function signature matching that attempts to define match as an isomorphism with or without unification. Rittri [Rit89, Rit92b, Rit92a] and DiCosmo [DC92] take this approach and have now identified four different matches, each time claiming to have found the right one.

Tools that manage the use of multiple definitions alleviate the problem of users being unable to decide which match to use. Such tools order the results by the closeness of the match (e.g., how many variable substitutions did we have to make) or set a range of acceptable number of hits and either relax or strengthen the match automatically until the number of components retrieved is within the range.

7.2 Future Work

Directions for future work include considering various ways of making both signature and specification matching more practical, exploring new applications of signature and specification matching, and applying our ideas to matching larger components. We consider future work first in signature matching (Section 7.2.1), then in specification matching (Section 7.2.2), work applicable to both signatures and specifications (Section 7.2.3), and approaches for matching larger components (Section 7.2.4).

7.2.1 Function Signature Matching

Practical Use

Function signature matching applies for any statically-typed programming language. If we built a signature-based retrieval tool for more commonly used languages and libraries, then we would be able to show the effectiveness of signature matching in a more widely accepted environment with larger libraries and more users. Examples of larger libraries include the C++ library from NIH [Gor87] or some of the growing number of software repositories available on the World Wide Web [BDGM95, PW95].

Given a real library and active users, we would then be able to do a variety of user studies to learn more about how signature matching is used in practice. Beagle, the signature-based retrieval tool for SML, includes the means to log what queries users made, which match relaxations they used, and which results they looked at further. Due to a limited library and user community, however, we did not have enough other users to reach any conclusions about its practicality. Implementing a tool for a larger library, publicizing the tool, and making it available on the World Wide Web should generate a lot more data about how a signature-based retrieval tool is used.

An alternative approach would be to gather a collection of programming tasks and then do a controlled comparative study that measures how long it takes users to complete the tasks with signature-based retrieval versus text-based retrieval.

For any retrieval system, an important factor is the way that relaxations are controlled and how the results are presented. With Beagle, we allow the user to control which relaxations are used for the match. As we discussed in Chapter 5, which relaxations are appropriate depends on the application. For statistical analysis, the user usually wants complete control over which relaxations are used. For retrieval for reuse, it may be more appropriate to focus on providing the user with a reasonable number of matches from which to choose. In this case, we could extend the system to add or remove relaxations until it gets a reasonable number of matches (if possible). For example, suppose a user specifies that he or she would like to choose from between five and 25 matches. Given a query, the system would start with the most relaxed match. If that retrieves more than 25 matches, the system would try again with all but one relaxation (using heuristics to determine which relaxation to remove). This process would repeat until the results are either within the range or as close as possible.

The main relaxations affecting the number of matches are usually generalized and specialized match. Another potential addition to a signature-based retrieval system would be to provide the user with more control over the way in which type variables are instantiated. In particular the user could specify that the instantiation of a variable should be limited to either just base types or to anything but functional types. Such a limitation would be useful in reducing unwanted matches. For example, we could extend the Analysis 1 example (pg. 74) by also finding out how many functions are instances of $\alpha \to \beta$ but not $\alpha \to \beta \to \gamma$ by requiring that β in the first query not be instantiated by a function type. Similarly, in Reuse 3 (pg. 72), we could eliminate the 24 library functions of type $\alpha \to \beta$ from the results of the query $\mathit{string\ list} \to ((\mathit{string} * \mathit{string}) \to \mathit{string}) \to \mathit{string}$ by limiting instantiation of type variables to non-functional types. Thus, instead of two useful functions out of 30 matches, we would find two useful functions out of only six matches.

Another approach to managing the results of a retrieval is to try to order the results of a retrieval so that the matches that are most likely to be useful are listed first. Other systems (e.g., Rittri [Rit92a]) have reported that in the presence of variable substitution or unification, ordering results by the number of substitutions required tended to put the most useful functions first. Their experience is consistent with our examples in Section 5.1.1 (e.g., Reuse 1 and Reuse 3), where using generalized match included not only useful functions (which in these cases contained only one variable) but also non-useful functions with the very general types $\alpha \to \beta$ and $\alpha \to \beta \to \gamma$ (hence two or three variables).

Signature Matching and Type Checking

Another direction for future work on signature matching is to consider other applications. One example would be to integrate function signature matching with a language's type checking system to provide additional feedback when a type checking error is found. Relaxed matches would detect problems with the order or format of arguments to a function, and retrieval would suggest other functions in cases where the programmer has gotten the name of the function wrong.

Consider the following scenario of how such a system would work. Suppose the code being checked contains the function application *foo* x, where $x : \tau_1$ and *(foo* x*)*: τ_2 by type inference, and *foo* has type $\tau_1' \rightarrow \tau_2'$. Suppose there is a type checking error because $\tau_1 \neq \tau_1'$ or $\tau_2 \neq \tau_2'$ (or because $\tau_1 \rightarrow \tau_2 \not\leq \tau_1' \rightarrow \tau_2'$ in a polymorphic system). With signature matching, we check whether $\tau_1' \rightarrow \tau_2'$ is matched by $\tau_1 \rightarrow \tau_2$ under relaxations reorder, uncurry, and type constructor. If they match, we print an error message suggesting how to reformat the input (e.g., "*foo (y,z)* does not type check, but *foo (z,y)* would"). Second, we use signature-based retrieval to search for functions in the environment that match the query $\tau_1 \rightarrow \tau_2$. If we find such a function *bar*, then in addition to the regular error message, we add a suggestion: "Perhaps you meant *bar* rather than *foo*," or we provide a way for the user to see a list of functions that matched $\tau_1 \rightarrow \tau_2$.

7.2.2 Function Specification Matching

Other Specifications

One direction for future work on function specification matching is to ask whether there is some middle ground between signature matching and matching of full-blown formal specifications. By constraining the specification language, we improve the tractability of the match. For example, rather than allowing unrestricted predicate logic terms in pre- and post-conditions, we could restrict the clauses to conjunctions of attribute/value pairs (e.g., **ensures** *operation* = *add* $\wedge$ *ordering* = *increasing*). The set of possible attribute and value terms would be defined in advance; there may be relations between values of a particular attribute, e.g., (*ordering* = *increasing*) $\Rightarrow$ (*ordering* = *nondecreasing*). Matching is a matter of checking a finite number of attributes to see whether their values match (or are related). Most of the same match definitions still apply, but the match is always decidable. An additional advantage to this approach is that the specifications might be easier for users to write and understand. Relaxing specifications to conjunctions of attribute/value pairs is similar to keyword text-based matching approaches (Section 6.3), but doing it within the framework of specification match might enable us to draw correspondences between attribute/value specifications and an underlying more formal specification so that we still get some of the same substitution behavior guarantees that we get with the formal specification matching approach.

Another example of reduced specification match specifies a function with a set of legal input/output pairs (like the query-by-example approach in Section 6.3). A function S can be substituted for a function Q if S's set of input/output pairs is a superset of Q's.

Better Practical Results

Another direction for future work on specification matching is to make it easier to prove the matches presented in Chapter 3. Many of the proofs that we did for the examples in Chapter 3 (10 out of 14) required at least some user guidance. With further experience, we might find that adding a few more default proof strategies would eliminate much of the user guidance. For example, three of the proofs that required guidance needed only the additional command **resume by induction**. If induction is added as a default proof strategy, those matches go through without additional guidance.

With further experience, we will also be able to determine how often in practice a component S matches with a component Q under weak post match but not under plug-in match or plug-in post match. Recall that weak post match allows us to assume that S_{pre} holds, thus allowing us to exclude cases for which we might not be able to show a relation between the post-conditions (e.g., the case where a container is empty on a delete operation).

One of the issues with specification matching is knowing which base terms to use in the specification clauses. Currently we assume that the two specifications we are attempting to match are based on the same LSL trait and thus use the same operator names. This is a reasonable assumption for applications like maintenance, where we are matching the specification of an upgraded component with the specification of the old component, and the specification of the old component was probably known to the specifier of the upgraded component. If we want to match two components that were specified independently, however, we will need to relate non-equivalent base terms. One way of identifying base terms that have different names but might intend the same thing is to use signature matching. Sorts would correspond to user-defined types and operators would correspond to functions. Matching would use the type constructor and reorder relaxations. So for example, an operator *add* : $C, E \to C$ in trait *Container1* would match an operator *insert* : $E, S \to S$ in trait *Container2*. Once signature matching has identified a correspondence between sorts and operators in the two traits, we would then have to prove that, under the correspondence, the theory of one trait contains the theory of the other (with the appropriate renamings), which we could do with LP [GGH90].

7.2.3 Signatures and Specifications

Automatic Programming

A potential application of both signature and specification matching is automatic program generation from library components. We would need to extend signature-based retrieval with a notion of a composite match. Given a query $\tau_1 \to \tau_2$, find two components $f1 : \tau_1 \to \tau_3$ and $f2 : \tau_3 \to \tau_2$ (or use relaxed signature matching so that the types need not match exactly).

Then use specification matching to check that the specification of $f1 \circ f2$ matches with the query specification. If necessary, recurse to get a chain of functions to be composed. Note that this is very similar to the planning task in AI [IJC95].

Mismatch

For both signature and specification matching our focus so far has been on when two components match. We could just as easily consider the question of when two components do not match. Depending on the components, identifying mismatches may be faster or easier than showing a match.

Pruning an indexed library is an example of using signature mismatch. Let τ_q be the query, τ_1 and τ_2 are in the indexed library, and τ_2 is a child of τ_1 (i.e., $\tau_1 > \tau_2$). Then if $\tau_q \geq \tau_1$, we know that $\tau_2 \not\geq \tau_q$ (i.e., there is a mismatch between τ_2 and τ_q), and hence we are able to prune τ_2 (and all other children of τ_1).

For specification matching, suppose we want to substitute S for Q, and Q modifies nothing. If S modifies anything at all, we cannot substitute and get the same observable behavior, so there is a mismatch between S and Q. Identifying other such mismatches might even make specification matching practical for retrieval.

7.2.4 Larger Components

In order to build really large systems, we need building blocks that are bigger than the functions and modules we have considered so far. The higher-level design of a software system is described by a software architecture [GS93], which consists of descriptions of the components of a system and of the way that components interact.

Components are like our modules in that they contain information about the functionality of the component (i.e., an interface). However, components also include information about how the component communicates with the rest of the world. Thus, there are two factors to consider in order to do signature or specification matching of these kinds of components: matching the interface and matching the communication protocols. We believe that the solution to the first problem is the use of a type system at the module level, as described in Section 4.4. There are formal descriptions of protocols and protocol matching [AG94]. Our approach would be to try to combine this notion of protocol match with signature or specification match (e.g., two components match if their specifications match and their protocols match). It is also possible that we could apply some aspects of our approach to matching to protocol matching. For example, we could consider whether there are any forms of relaxed protocol matches.

There are two different kinds of match to consider. First, matching to determine when we can replace one component with another (such as the matches we have considered in this thesis). A second kind of match is to determine when two components can be connected together. This is the kind of match that other work on interoperability has focused on [AG94, WWRT91, Kon93, YS94, Tha94]. The same techniques apply to both, but the actual match definitions are different. In the first case, the functionality of the two components should be similar; in the second case, the results (e.g., output) of the first component should be compatible with the expectations (e.g., input) of the second component. Both kinds of match are important and both are amenable to signature and specification matching and to checking for mismatch.

7.3 Epilogue

We envision a world where semantic abstracts and semantic-based matching provide tools that greatly improve the tasks of creating and managing software. At the function level, a retrieval system like Beagle should be a part of every software development environment. At higher levels, we imagine the development of richer type systems to capture the important abstractions about components. Moving beyond software components, we use the same approach to address the issue of managing the increasing amounts of information: type systems and specifications to describe the data, various notions of relaxed matches to compare the data, and tools based on the matches to help us manage the data.

Appendix A

The Container Trait

The *Container* trait defines operators to generate containers (*empty* and *insert*), to return the element or container resulting from deleting an element from the beginning or end (*first*, *last*, *butFirst*, and *butLast*), to return the length of a container (*length*), and to determine whether a container is empty (*isEmpty*).

$Container(E, C)$: **trait**
  includes *Integer*
  introduces

$empty : \to C$	$butFirst : C \to C$
$insert : E, C \to C$	$butLast : C \to C$
$first : C \to E$	$isEmpty : C \to Bool$
$last : C \to E$	$length : C \to Int$

  asserts
    C **generated by** *empty*, *insert*
    C **partitioned by** *isEmpty*, *length*
    $\forall\ e : E, c : C$
      $first(insert(e, c)) == e$
      $butFirst(insert(e, c)) == c$
      $last(insert(e, c)) == \textbf{if}\ c = empty\ \textbf{then}\ e\ \textbf{else}\ last(c)$
      $butLast(insert(e, c)) == \textbf{if}\ c = empty\ \textbf{then}\ empty\ \textbf{else}\ insert(e, butLast(c))$
      $isEmpty(empty)$
      $\neg isEmpty(insert(e, c))$
      $length(empty) == 0$
      $length(insert(e, c)) == length(c) + 1$

Appendix B

Subtype Specification

Figure B.1 shows the *Container2* trait on which object specifications *BagObj* and *StackObj* are based. This specification is different from the *Container* trait in Appendix A in that it has additional operators *delete*, *isIn*, and *count*. Figures B.2 and B.3 show the results of translating the specifications to LP input.

Container2 (E , C) : **trait**
  includes *Integer*
  introduces

$empty : \to C$	$butFirst : C \to C$
$insert : E, C \to C$	$butLast : C \to C$
$delete : E, C \to C$	$isEmpty : C \to Bool$
$first : C \to E$	$isIn : E, C \to Bool$
$last : C \to E$	$count : E, C \to Int$
$size : C \to Int$	

  asserts
    C **generated by** *empty*, *insert*
    C **partitioned by** *count*
    $\forall\ e, e1 : E, c : C$
      $last(insert(e, c)) == e$
      $butLast(insert(e, c)) == c$
      $first(insert(e, c)) ==$ **if** $c = empty$ **then** e **else** $first(c)$
      $butFirst(insert(e, c)) ==$ **if** $c = empty$ **then** $empty$ **else** $insert(e, butFirst(c))$
      $isEmpty(empty)$
      $\neg isEmpty(insert(e, c))$
      $\neg isIn(e, empty)$
      $isIn(e, insert(e1, c)) == (e = e1) \vee (isIn(e, c))$
      $size(empty) == 0$
      $size(insert(e, c)) == size(c) + 1$
      $count(e, empty) == 0$
      $count(e, insert(e1, c)) == count(e, c) +$ (**if** $e = e1$ **then** 1 **else** 0)
      $count(e, delete(e1, c)) ==$ **if** $e = e1$ **then** $max(0, count(e, c) - 1)$ **else** $count(e, c)$

Figure B.1: *Container2* trait

```
%% signature BagObj
set name BagObj

%% type 'a t based on Container2.E Container2.C

%% Variable declarations
declare var
  b : C
  b' : C
  e: E
  n: Int
  ..

%% Specification declarations
declare op
  putPre: -> Bool
  putPost: C, C, E -> Bool
  getPre: C, E-> Bool
  getPost: C, C, E-> Bool
  cardPre: -> Bool
  cardPost: C, Int-> Bool
  ..

%% Specification assertions
assert
  putPre = true;
  putPost(b, b', e) = (b' = insert(e,b));
  getPre(b, e) = (~(isEmpty(b)));
  getPost(b, b', e) = (b' = delete(e,b) /\ isIn(e,b));
  cardPre = true;
  cardPost(b, n) = (n = size(b))
  ..
```

Figure B.2: Bag specification translated to LP input

```
%% signature StackObj
set name StackObj

%% type 'a t based on Container2.E Container2.C

%% Variable declarations
declare var
  s : C
  s' : C
  e: E
  i: Int
  ..

%% Specification declarations
declare op
  pushPre: -> Bool
  pushPost: C, C, E -> Bool
  popPre: C, E-> Bool
  popPost: C, C, E-> Bool
  swap_topPre: C, E -> Bool
  swap_topPost: C, C, E -> Bool
  heightPre: -> Bool
  heightPost: C, Int-> Bool
  ..

%% Specification assertions
assert
  pushPre = true;
  pushPost(s, s', e) = (s' = insert(e,s));
  popPre(s, e) = (~(isEmpty(s)));
  popPost(s, s', e) = (s' = butLast(s) /\ e = last(s));
  swap_topPre(s, e) = (~(isEmpty(s)));
  swap_topPost(s, s', e) = (s' = insert(e,butLast(s)));
  heightPre = true;
  heightPost(s, i) = (i = size(s))
  ..
```

Figure B.3: Stack specification translated to LP input

Bibliography

[AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.

[AM87] William W. Agresti and Frank E. McGarry. The Minnowbrook workshop on software reuse: A summary report. In Will Tracz, editor, *Tutorial: Software Reuse: Emerging Technology*, pages 33–40. Computer Society Press, 1987.

[Ame91] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *LNCS*, pages 60–90. Springer-Verlag, NY, 1991.

[AS87] Susan P. Arnold and Stephen L. Stepoway. The REUSE system: Cataloging and retrieval of reusable software. In Will Tracz, editor, *Tutorial: Software Reuse: Emerging Technology*, pages 138–141. Computer Society Press, 1987.

[ATT93] The Standard ML of New Jersey library reference manual. Technical report, AT&T Bell Laboratories, February 1993.

[BDGM95] Shirley Brown, Jack Dongarra, Stan Green, and Keith Moore. Location-independent naming for virtual distributed software repositories. In *Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'95)*, pages 179–185, April 1995.

[Ber91] Dave Berry. The Edinburgh SML library. Technical Report ECS-LFCS-91-148, University of Edinburgh, April 1991.

[Bis92] Walter R. Bischofberger. Sniff – a pragmatic approach to a C++ programming environment. In *USENIX C++ Conference*, pages 67–81, August 1992.

[BP89] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability Vol. 1: Concepts and Models*. ACM Press, N.Y., 1989.

[Car89] Luca Cardelli. Typeful programming. Report 45, DEC Systems Research Center, Palo Alto, CA, May 1989.

[CE93] Yuk Fung Chang and Caroline M. Eastman. An information retrieval system for reusable software. *Information Processing and Management*, 29(5):601–614, 1993.

[CHJ93] P. S. Chen, R. Hennicker, and M. Jarke. On the retrieval of reusable software components. In *Proceedings of the 2^{nd} International Workshop on Software Reusability*, pages 99–108. IEEE Computer Society Press, March 1993.

[CMR92] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *Proceedings of the 14^{th} International Conference on Software Engineering*, pages 138–156, May 1992.

[Cor95] InfoSeek Corporation. Infoseek home page. Santa Clara, California. http://www.infoseek.com, 1995.

[DC92] Roberto Di Cosmo. Type isomorphisms in a type-assignment framework. In *Proceedings of the 19^{th} Annual Symposium on Principles of Programming Languages*, pages 200–210, January 1992.

[DL92] Krishna Kishore Dhara and Gary T. Leavens. Subtyping for mutable types in object-oriented programming languages. Technical Report 92-36, Department of Computer Science, Iowa State University, Ames, Iowa, November 1992.

[FF93] M. G. Fugini and S. Faustle. Retrieval of reusable components in a development information system. In *Proceedings of the 2^{nd} International Workshop on Software Reusability*, pages 89–98. IEEE Computer Society Press, March 1993.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[FHR91] Gerhard Fischer, Scott Henninger, and David Redmiles. Cognitive tools for locating and comprehending software objects for reuse. In *Proceedings of the 13^{th} International Conference on Software Engineering*, May 1991.

[FKS94] B. Fischer, M. Kievernagel, and W. Struckmann. VCR: A VDM-based software component retrieval tool. Technical Report 94-08, Technical University of Braunschweig, Germany, November 1994.

[FN87] W. B. Frakes and B. A. Nejmeh. Software reuse through information retrieval. In *Proceedings of the 20^{th} Annual Hawaii International Conference on System Sciences (HICSS), Volume II*, pages 530–534, 1987.

[GG91] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Report 82, DEC Systems Research Center, Palo Alto, CA, December 1991.

[GGH90] Stephen J. Garland, John V. Guttag, and James J. Horning. Debugging Larch Shared Language specifications. Report 60, DEC Systems Research Center, Palo Alto, CA, July 1990.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[Gor87] Keith E. Gorlen. An object-oriented class library for C++ programs. *Software – Practice and Experience*, 17(12):899–922, December 1987.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering, Volume 1.* World Scientific Publishing Company, N.J., 1993.

[Hal93] Robert J. Hall. Generalized behavior-based retrieval. In 15^{th} *International Conference on Software Engineering*, pages 371–380, 1993.

[Hen95] Scott Henninger. Supporting the process of satisfying information needs with reusable libraries: An empirical study. In *Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'95)*, pages 267–270, April 1995.

[HM91] Richard Helm and Yoëlle S. Maarek. Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries. In *OOPSLA Conference Proceedings*, pages 47–61, 1991.

[IEE84] IEEE Transactions on Software Engineering, September 1984. SE-10(5).

[IJC95] Working notes, IJCAI workshop on formal approches to the reuse of plans, proofs, and programs, August 1995.

[JC92] J.-J. Jeng and B. H. C. Cheng. Formal methods applied to reuse. In *Proceedings of the* 5^{th} *Workshop in Software Reuse*, 1992.

[JC95] J.-J. Jeng and B. H. C. Cheng. Specification matching for software reuse: A foundation. In *Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'95)*, pages 97 –105, April 1995.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International, 1986.

[Kon93] Dimitri Konstantas. Object-oriented interoperability. In Oscar M. Nierstrasz, editor, *ECOOP'93 – 7^{th} European Conference on Object-Oriented Programming, Kaiserslautern, Germany, July 1993*, volume 707 of *LNCS*, pages 80–102. Springer-Verlag, NY, 1993.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[KRT87] Shmuel Katz, Charles A. Richter, and Khe-Sing The. PARIS: A system for reusing partially interpreted schemas. In *Proceedings of the 9^{th} International Conference on Software Engineering*, pages 377–385, March 1987.

[Kru92] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[KYS85] Hideki Katoh, Hiroyuki Yoshida, and Masakatsu Sugimoto. Logic-based retrieval and reuse of software. Technical Report TR-153, Institute for New Generation Computer Technology, October 1985.

[Lea89] Gary Leavens. Verifying object-oriented programs that use subtypes. Technical Report 439, MIT Laboratory for Computer Science, February 1989. Ph.D. thesis.

[Ler95a] Xavier Leroy. CAML light manual. Technical report, INRIA, July 1995.

[Ler95b] Xavier Leroy. The Caml Special Light system, release 1.07, documentation and user's manual. Technical report, INRIA, September 1995.

[Lis87] Barbara Liskov. Data abstraction and hierarchy. In *OOPSLA '87: Addendum to the Proceedings*, pages 17–34, 1987.

[LW90] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes. In *ECOOP/OOPSLA '90 Proceedings*, 1990.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.

[MBK91] Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 8(17):800–813, August 1991.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, 1988.

[MF93] Jean-Marc Morel and Jean Faget. The REBOOT enviornment. In *Proceedings of the 2^{nd} International Workshop on Software Reusability*, pages 80–88. IEEE Computer Society Press, March 1993.

[MFCS95] Eliseo Mambella, Roberto Ferrari, Francesca De Carli, and Angela Lo Surdo. An integrated approach to software reuse practice. In *Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'95)*, pages 63–71, April 1995.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[ML94] M. Mauldin and J. Leavitt. Web-agent related research at the CMT. In *ACM Special Interest Group on Networked Information Discovery and Retrieval (SIGNIDR-94)*, August 1994.

[MMM94] A. Mili, R. Mili, and R. Mittermeir. Storing and retrieving software components: A refinement-based approach. In *Proceedings of the* 16^{th} *International Conference on Software Engineering*, May 1994.

[MMM95] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, June 1995.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[OHPDB92] Eduardo Ostertag, James Hendler, Rubén Prieto-Díaz, and Christine Braun. Computing similarity in a reuse library system: An AI-based approach. *ACM Transactions on Software Engineering and Methodology*, 1(3):205–228, July 1992.

[PD89] Rubén Prieto-Díaz. Classification of reusable modules. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability Vol. 1: Concepts and Models*, pages 99–123. ACM Press, N.Y., 1989.

[Per89] Dewayne E. Perry. The Inscape environment. In *Proceedings of the* 11^{th} *International Conference on Software Engineering*, pages 2–12, 1989.

[Pou93] Jeffery S. Poulin. Integrated support for software reuse in computer-aided software engineering (CASE). *ACM SIGSOFT Software Engineering Notes*, 18(4):75–82, October 1993.

[PP92] Andy Podgurski and Lynn Pierce. Behavior sampling: A technique for automated retrieval of reusable components. In 14^{th} *International Conference on Software Engineering*, pages 349 – 360, 1992.

[PP93] Dewayne E. Perry and Steven S. Popovich. Inquire: Predicate-based use and reuse. In *Proceedings of the* 8^{th} *Knowledge-Based Software Engineering Conference*, pages 144–151, September 1993.

[PP94] Santanu Paul and Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 6(20):463–475, June 1994.

[PW95] Jeffery S. Poulin and Keith J. Werkman. Melding structured abstracts and the world wide web for retrieval of reusable components. In *Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'95)*, pages 160–168, April 1995.

[Rit89] Mikael Rittri. Using types as search keys in function libraries. *Conference on Functional Programming Languages and Computer Architectures*, pages 174–183, September 1989.

[Rit90] Mikael Rittri. Retrieving library identifiers via equational matching of types. In 10^{th} *International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Number 449, pages 603–617. Springer-Verlag, July 1990.

[Rit91] Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, January 1991.

[Rit92a] Mikael Rittri. Retrieving library functions by unifying types modulo linear isomorphism. Technical Report 66, Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, 1992.

[Rit92b] Mikael Rittri. Retrieving library identifiers via equational matching of types. Technical Report 65, Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, May 1992.

[RT89] Colin Runciman and Ian Toyn. Retrieving re-usable software components by polymorphic type. *Conference on Functional Programming Languages and Computer Architectures*, pages 166–173, September 1989.

[RT91] Colin Runciman and Ian Toyn. Retrieving reusable software components by polymorphic type. *Journal of Functional Programming*, 1(2):191–211, April 1991.

[RW91] Eugene J. Rollins and Jeannette M. Wing. Specifications as search keys for software libraries. In *Proceedings of the Eighth International Conference on Logic Programming*, June 1991.

[SC94] David W.J. Stringer-Calvert. Signature matching for Ada software reuse. Master's thesis, University of York, 1994.

[SGS91] Gregor Snelting, Franz-Josef Grosch, and Ulrik Schroeder. Inference-based support for programming in the large. In A. van Lamsweerde and A. Fugetta, editors, 3^{rd} *European Software Engineering Conference*, number 550 in Lecture Notes in Computer Science, pages 396–408. Springer Verlag, October 1991.

[SM83] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[Spi88] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge University Press, 1988.

[SSS93] Lars Sivert Sørumgård, Guttorm Sindre, and Frode Stokke. Experiences from application of a faceted classification scheme. In *Proceedings of the* 2^{nd} *International Workshop on Software Reusability*, pages 116–124. IEEE Computer Society Press, March 1993.

[Sta86] Richard Stallman. *GNU Emacs Manual*, 1986.

[Ste84] Guy L. Steele Jr. *Common Lisp, The Language.* Digital Press, 1984.

[Tes81] Larry Tessler. The Smalltalk environment. *BYTE*, pages 90–147, August 1981.

[Tha94] Satish R. Thatté. Automated synthesis of interface adapters for reusable classes. In *Proceedings of the* 21^{st} *Annual Symposium on Principles of Programming Languages*, pages 174–187, January 1994.

[TR93] David Tarditi and Gene Rollins. Local guide to Standard ML. Technical report, CMU, March 1993.

[Wad89] Philip Wadler. Theorems for free! In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, September 1989.

[WRZ93] J.M. Wing, E. Rollins, and A. Moormann Zaremski. Thoughts on a Larch/ML and a new application for LP. In Ursula Martin and Jeannette M. Wing, editors, *First International Workshop on Larch.* Springer Verlag, 1993.

[WWRT91] Jack C. Wileden, Alexander L. Wolf, William R. Rosenblatt, and Peri L. Tarr. Specification-level interoperability. *CACM*, 34(5):72–87, May 1991.

[YS94] Daniel M. Yellin and Robert E. Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. *OOPSLA Conference Proceedings, ACM SIGPLAN Notices*, 29(10):176–190, October 1994.

[ZW93] Amy Moormann Zaremski and Jeannette M. Wing. Signature Matching: A Key to Reuse. In *Proceedings of SIGSOFT'93 First ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM SIGSOFT Software Engineering Notes,* **18**(5), pages 182–190, December 1993. Also CMU-CS-93-151, May, 1993.

[ZW95a] Amy Moormann Zaremski and Jeannette M. Wing. Signature Matching: a Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, April 1995.

[ZW95b] Amy Moormann Zaremski and Jeannette M. Wing. Specification Matching of Software Components. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM SIGSOFT Software Engineering Notes,* **20**(4), pages 6–17, October 1995. Also CMU-CS-95-127, March, 1995.