Signature Matching: a Tool for Using Software Libraries

AMY MOORMANN ZAREMSKI and JEANNETTE M. WING
Carnegie Mellon University

Signature matching is a method for organizing, navigating through, and retrieving from software libraries. We consider two kinds of software library components, functions and modules, and hence two kinds of matching, function matching and module matching. The signature of a function is simply its type; the signature of a module is a multiset of user-defined types and a multiset of function signatures. For both functions and modules, we consider not just *exact* match, but also various flavors of *relaxed* match. We describe various applications of signature matching as a tool for using software libraries, inspired by the use of our implementation of a function signature matcher written in Standard ML.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*Software Libraries; Modules and Interfaces*; D.2.m [**Software Engineering**]: Miscellaneous—*Reusable Software*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data Types and Structures*

General Terms: Design

Additional Key Words and Phrases: Software Retrieval; Signature Matching

1. WHAT IS SIGNATURE MATCHING?

Software libraries are a great resource for the software engineer. An engineer can study library components to become more familiar with a language or with a style of programming, look for common patterns of usage, and reuse components rather than writing code from scratch [Agresti and McGarry 1987; Biggerstaff and Perlis 1989; Ramamoorthy 1984; Fontana and Neath 1991]. One of the keys to using soft-

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.
The 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.
The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. This manuscript is submitted for publication with the understanding that the U. S. Government is authorized to reproduce and distribute reprints for Governmental purposes.
Authors' address: School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; email: amy;wing@cs.cmu.edu.
To appear, ACM Transactions on Software Engineering and Methodology (TOSEM), April 1995.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1995 ACM xxxx-xxxx/xx/xxxx-xxxx $xx.xx

ware libraries successfully, especially as libraries increase in size, is the availability of good tools to organize, navigate through, and retrieve from libraries.

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/`); one uses `ls` and `grep` to locate desired components. Aside from some of the information that could be gleaned from how the library is organized, the task of finding something in these libraries must rely on the names of components.

Software components, however, have additional built-in information with which a software engineer is familiar and upon which we can build additional tools for software libraries. For example, libraries for object-oriented languages have a hierarchical class inheritance structure which can be navigated by a browser (e.g., Smalltalk [Tesler 1981] and C++ [Bischofberger 1992]).

Signature matching uses another kind of built-in information: type information about the components. This paper defines signature matching in detail and shows how it can be used for various tasks on software libraries. Since many of our examples focus on the task of retrieving from a library and our work has foundations in that area, we use terminology from information retrieval to describe signature matching. To illustrate our ideas here and for the rest of this paper, consider the small library of components in Figure 1. It contains three ML signature modules, *LIST*, *QUEUE*, and *SET*, which together define seventeen functions, e.g., *empty* and *cons*. (ML *signature* modules are akin to Ada *definition* modules and Modula-3 *interface* modules; ML implementations are written in modules called *structures* [Milner et al. 1990].)

If we are looking for a specific function, rather than trying to guess its name, we could use the function's *type*, which is the list of types of its input and output parameters (and possibly information about what exceptions may be signaled). For example, α *list* $\rightarrow \alpha$ is the type of the function *hd*. If we are looking for a module, we could use its *interface*, which is a multiset of user-defined types and a multiset of function types. For example, *SET* has one user-defined type, α *T*, and seven function types. The *signature* of a component is exactly this information: the type of a function or the interface of a module. In practice, a library of software components is usually a set of program modules; we can construct a function library from a module library by extracting all functions from each module in the obvious way. We can reasonably assume that signature information is either provided with or derivable from code components, since this information is typically required by the compiler.

Signature matching is the process of determining which library components "match" a query signature. As with other information retrieval methods, requiring a component to match a query exactly will sometimes be too strong. There may be a component that does not match exactly, but is similar in some way and hence would match a query if the component (or query) is slightly modified. Thus, in addition to *exact match*, we also consider cases of *relaxed matches* between a query and a library component. The expectation is that relaxed matching returns components that are "close enough" to be useful to the software developer. For example, relaxed matching on functions might allow reordering of a library function's input

```
signature LIST =
  sig
    val empty : unit → α list
    val cons : (α, α list) → α list
    val hd : α list → α
    val tl : α list → α list
    val map : (α → β) → α list → β list
    val intsort : ((int, int) → bool) → int list → int list
  end
```

```
signature QUEUE =
  sig
    type α T
    val create : unit → α T
    val enq : (α, α T) → α T
    val deq : α T → (α, α T)
    val len : α T → int
  end
```

```
signature SET =
  sig
    type α T
    val create : unit → α T
    val insert : (α T, α) → α T
    val delete : (α T, α) → α T
    val member : (α, α T) → bool
    val union : (α T, α T) → α T
    val intersection : (α T, α T) → α T
    val difference : (α T, α T) → α T
  end
```

Fig. 1. Three ML Signature Modules

parameters; relaxed matching on modules might require only a subset of the library module's functions. We define signature matching in its most general form as follows:

Definition 1.1 (*Signature Match*).

Signature Match: Query Signature, Match Predicate, Component Library → Set of Components
Signature Match$(q, M, C) = \{c \in C : M(c, q)\}$.

In other words, given a signature query, q, a match predicate, M, and a library of components, C, signature matching returns a set of components, each of which satisfies the match predicate. This paper explores the design space of signature matching: we consider two kinds of library components, functions and modules, and hence consider two kinds of signature match, *function match* and *module match*. We are interested in both levels of signature match because in practice we expect users to retrieve at different levels of granularity. We also consider different kinds of match predicates: *exact match* and various *relaxed matches* (for both function and module match).

Signature matching gives us two ways to build tools for software libraries: using a query to describe and retrieve a subset of library components, and defining structures to index the library. The subset returned by signature matching against a particular query can be used to filter components for another tool, analyze or browse the library, or locate a component for reuse. The equivalence classes and partial orders defined over the library by match definitions can be used to index and navigate through the library. We view signature matching as complementing

standard search and browsing facilities, e.g., `grep` and `ls`, which provide a primitive means of accomplishing the same goals. A tool that does signature matching is just one of many in a software developer's environment. Using a signature matcher should be just as easy as doing a search on a string pattern.

Signature matching can be viewed as an instance of using domain-specific information to aid in the search process. Knowing that we are searching program modules as opposed to uninterpreted Unix files or SQL database records lets us exploit the structure and meaning of these components. Using domain-specific information is an idea applicable to other large information databases, e.g., the nationwide Library of Congress, law briefs, police records, geological maps, and may prove to be key in grappling with the problem of scale.

We define module match in terms of function match. So we begin at the lowest level of granularity in Section 2 by defining exact match and several relaxed matches for functions. Section 3 defines module match and its relaxations. In Section 4 we discuss the various applications of signature matching in more detail and describe our signature matching facility. We compare our work with other approaches in Section 5 and close with a summary and suggestions for future work in Section 6.

2. FUNCTION MATCHING

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 [1988]. A *type* is either a type *variable* $\in$ *TypeVar* (denoted by Greek letters) or a type *operator* $\in$ *TypeOp* applied to other types. Type operators are either built-in operators (*BuiltInOp*) or user-defined operators (*UserOp*). Each type operator has an *arity* indicating the number of type arguments. *Base types* are operators 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 operators (e.g., *int list* stands for the "list of integers" type). The user-defined type, $\alpha\ T$, represents a type operator T with arity 1, where the type of the argument to T is α.[1] In general, when we refer to type operators, they can be either built-in or user-defined (i.e., $BuiltInOp \cup UserOp$). Two types τ and τ' are *equal* ($\tau =_T \tau'$) if (1) they are lexically identical type variables or (2) $\tau = typeOp(\tau_1, ..., \tau_n)$, $\tau' = typeOp'(\tau'_1, ..., \tau'_n)$, $typeOp = typeOp'$, and $\forall\ 1 \leq i \leq n, \tau_i =_T \tau'_i$. *Polymorphic* types contain at least one type variable; types that do not contain any type variables are *monomorphic*.

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 the case where τ' is just a variable, $[\tau'/\alpha]\tau$ is simply *variable renaming*. For variable renaming, $\alpha, \tau' \in$ *TypeVar* or $\alpha, \tau' \in$ *UserOp*. We think of user-defined

[1] We deviate from ML's convention of using $*$ for tuple construction; the comma is easier on the eyes. Also, in ML, the common programming practice is to use T for the operator name of the user-defined type of interest.

operator names as variables for the purposes of renaming, since different users may use a different name for the same type operator. We do not allow renaming of built-in operators, since we assume that when users include a built-in operator in a query, they want exactly that type operator. Renaming sequences may include both type variable renaming and user-defined type operator renaming: $[\beta/\alpha][C/T](\alpha\ T \rightarrow \alpha) = (\beta\ C \rightarrow \beta)$. We will use V for a sequence of variable renamings and U for a sequence of more general substitutions.

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

Definition 2.1 (*Generic Function Match*).

M: Library Type, Query Type $\rightarrow$ Boolean
$M(\tau_l, \tau_q) = \exists\ T_l$ and T_q such that $T_l(\tau_l)\ R\ T_q(\tau_q)$

where the implicit parameter R is some relationship between types (e.g., equality) and T_l and T_q are transformations (e.g., reordering) that are applied to the library and query types, respectively. 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 R is the type equality ($=_T$) relation.

We classify relaxed function matches as either *partial* matches, which vary R, the relationship between τ_l and τ_q (e.g., define R to be a partial order), or *transformation* matches, which vary T_l or T_q, the transformations on types. In the following subsections, we first define exact match, followed by partial matches, transformation matches, and combined matches. Each of these match predicates can be used to instantiate the M of *Signature Match*; see Definition 1.1.

2.1 Exact Match

Definition 2.1.1 (*Exact Match*).

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

Two function types match exactly if they match modulo variable renaming. Recall that variable renaming may rename either type variables or user-defined type operators. For monomorphic types with no user-defined 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 $\tau_q = (\beta, \beta) \rightarrow bool$ with the substitution $V = [\beta/\alpha]$. But $\tau_l = \alpha \rightarrow \beta$ and $\tau_q = \gamma \rightarrow \gamma$ do not match 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 useful in practice, consider two examples where the library is the set of all functions in Figure 1. Suppose a user wants to locate a function that applies an input function to each element of a list, forming a new list. The query $\tau_q = (\alpha \rightarrow \gamma) \rightarrow \alpha\ list \rightarrow \gamma\ list$ matches the *map* function (with the renaming $[\gamma/\beta]$), exactly what the user wants. As a second example, suppose a user wants to locate a function to add an element to a collection with the query $\tau_q = (\alpha, \alpha\ C) \rightarrow \alpha\ C$. This query retrieves the *enq* function on queues (with the renaming $[C/T]$), which may be what the user wants, but not the *insert* function on sets, another likely candidate.

2.2 Partial 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.

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 Definition 2.1 of generic function match, for exact match, the relation, 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.

Definition 2.2.1 (*Generalized Match*).

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

A library type matches 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 $\tau_q = (int \rightarrow bool) \rightarrow int\ list \rightarrow bool\ list$. This query does not match exactly with any function in our library. But a generalized match would return *map* for this query, 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.

Definition 2.2.2 (*Specialized Match*).

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

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

As an example of how specialized match can be 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 specialized match would return *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.

Although we present generalized and specialized match in terms of changing the relation (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 (Alternative) 2.2.3 (*Generalized and Specialized Match*).

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

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

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 kinds of relaxed matches (Section 2.4).

Note that *type unification* [Field and Harrison 1988] is just a combination of generalized and specialized match, allowing the variable substitutions to occur on either type.

Definition 2.2.4 (*Unify Match*).

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

In practice, we do not expect unify match to be of as much use as either generalized or specialized match, since the relation between types τ_q and τ_l is more complicated with unification. However, it is important to relate type unification and type matching, i.e., the former is definable in terms of the latter.

2.3 Transformation Relaxations

Other kinds of relaxed match on functions *transform* a type expression to achieve a match. Examples include 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 the same since we can view multiple arguments to a function as a tuple. For example, 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, and $\tau_q = (\alpha\ list, \alpha) \rightarrow \alpha\ list$ would miss *cons* because the types in the tuple are in a different order.

2.3.1 *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^*:

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

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 2.3.1 (*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. In the exact-match example at the end of Section 2.1, we were lucky and guessed the "right" query $((\alpha, \alpha\ C) \rightarrow \alpha\ C)$ to match exactly with *enq*. This time, suppose we use the query $\tau_q = \alpha\ C \rightarrow \alpha \rightarrow \alpha\ C$. This query does not match exactly with any functions in our library, but uncurry match would return the functions *insert* and *delete* on a set. Note that this query does not match *enq*, the other likely candidate.

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, $match_{curry}(\tau_l, \tau_q) = match_E(curry(\tau_l), curry(\tau_q))$. Note that $match_{curry}(\tau_l, \tau_q)$ if and only if $match_{uncurry}(\tau_l, \tau_q)$.

2.3.2 *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 2.3.2 (*Reorder Match*).

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

Under this relaxation, a library type, τ_l, matches 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 2.3 now matches with 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 operators, e.g., so that $(int, \alpha)\ T \rightarrow int$ and $(\alpha, int)\ T \rightarrow int$ would match.

2.4 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. Intuitively, 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) $\alpha\ T \rightarrow \alpha \rightarrow \alpha\ T$

Reorder match allows a query of type 1 or 2 to match library functions of types 1 or 2, but not types 3 or 4. Uncurry match allows a query of type 1 or 3 to match library functions of those types (and likewise for types 2 and 4). But no individual relaxed match allows a single query to match all four types. By composing reorder and uncurry match, a query of any of the four types will match a 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 $match_{spec}$, each of the relaxed match definitions presented in Sections 2.2 and 2.3 can be cast in a composable form by instantiating R to $match_E$ in Definition 2.1, the generic function match definition:

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

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

Definition 2.4.1 (*Match Composition*).

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

The choice of $R1$ determines the kinds of transformations in $T1$ (and likewise for $R2$ and $T2$). For $match_{gen}$, T_l is a sequence of variable substitutions 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 simplicity, we omit the recursive versions of $match_{uncurry^*}$ and $match_{reorder^*}$, although the analysis below could be easily extended to include them. Since $match_{spec}$ and $match_{unify}$ can be defined in terms of $match_{gen}$,[2] we also exclude them from our analysis. Thus, there are three "basic" relaxed matches: $match_{gen}$, $match_{uncurry}$, and $match_{reorder}$. That is, $R1, R2 \in \{$gen, uncurry, reorder $\}$.

We can compose any number of relaxed matches in any order. The order in which they are composed does make a difference; transformations are not generally commutative. For example, $match_{gen}$ does not commute with either $match_{uncurry}$ or $match_{reorder}$ because in either case, the variable substitution from generalizing could introduce a type that could then be transformed by uncurrying or reordering, but would not be transformed if the variable substitution is done last. Suppose $\tau_q = (bool, int) \rightarrow (int, bool)$ and $\tau_l = \alpha \rightarrow \alpha$. $(match_{gen} \circ\ match_{reorder})(\tau_l, \tau_q)$ is false, but $(match_{reorder} \circ\ match_{gen})(\tau_l, \tau_q)$ is true with the substitution $[(int, bool)/\alpha]$

[2] $match_{spec}(\tau_l, \tau_q) = match_{gen}(\tau_q, \tau_l)$, and $match_{unify}(\tau_l, \tau_q) = (match_{gen} \circ match_{spec})(\tau_l, \tau_q) = (match_{spec} \circ match_{gen})(\tau_l, \tau_q)$

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 into type expressions that contain a tuple.

We now consider some of the interesting combinations of these relaxed matches, those we expect to be useful in practice. The relations between the various combinations of relaxed matches lead to a natural partial ordering relation on combined matches, based on the set of function types that a match defines (namely, the set of types that match a given query type).

—$(match_{reorder} \circ match_{uncurry})(\tau_l, \tau_q)$.
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 $\tau_q = \alpha \rightarrow \alpha\ T \rightarrow \alpha\ T$ would match *enq* ($\tau_l = (\alpha, \alpha\ T) \rightarrow \alpha\ T$) on queues and *insert* and *delete* ($\tau_l = (\alpha\ T, \alpha) \rightarrow \alpha\ T$) on sets.

—$(match_{gen} \circ match_{uncurry})(\tau_l, \tau_q)$.
τ_l and τ_q match if the uncurried form of τ_l is more general than the uncurried form of τ_q. With this composition, the query $\tau_q = ((int \rightarrow bool), int\ list) \rightarrow bool\ list$ would match the *map* function ($\tau_l = (\alpha \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta\ list$).

—$(match_{gen} \circ match_{reorder})(\tau_l, \tau_q)$.
τ_l and τ_q match if some permutation of τ_l is more general than τ_q. With this composition, the query $\tau_q = (int\ T, int) \rightarrow bool$ would match the *member* function on sets $\tau_l = (\alpha, \alpha\ T) \rightarrow bool$.

—$(match_{gen} \circ match_{reorder} \circ match_{uncurry})(\tau_l, \tau_q)$.
τ_l and τ_q match if some permutation of the uncurried form of τ_l is more general than the uncurried form of τ_q. Using this combined match with the order of τ_l and τ_q reversed (i.e., using $match_{spec}$ instead of $match_{gen}$), with the query $\tau_q = (\alpha\ list, (\alpha, \alpha \rightarrow bool)) \rightarrow \alpha\ list$ matches the *intsort* function in our library ($\tau_l = (int, int \rightarrow bool) \rightarrow int\ list \rightarrow int\ list$).

2.5 Generic Match

As we mentioned at the beginning of this section, all of the function matches fit the same generic match definition: $M(\tau_l, \tau_q) = \exists\ T_l$ and T_q such that $T_l(\tau_l)\ R\ T_q(\tau_q)$. Table I shows how 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 two of the combined matches to show how the matches can be combined. The relation 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 (renaming), U (substitution), T_σ (permute tuple) or UC (uncurry).

3. MODULE MATCHING

Function matching addresses the problem of locating a particular function in a component library. A programmer, however, often needs a collection of functions, e.g., one that provides a set of operations on an abstract data type. Most modern programming languages explicitly support the definition of abstract data types through a separate modules facility, e.g., CLU clusters, Ada packages, or C++

Match	R	T_l	T_q
Exact	$=_T$	V	Id
Generalized	$\geq$	Id	Id
Generalized (alternative)	$=_T$	U	Id
Specialized	$\leq$	Id	Id
Specialized (alternative)	$=_T$	Id	U
Unify	$=_T$	U	U
Uncurry	$match_E$	UC	UC
Reorder	$match_E$	T_σ	Id
Reorder ∘ Uncurry	$match_E$	$T_\sigma \circ UC$	UC
Generalized ∘ Uncurry	$\geq$	UC	UC

Table I. Instantiations of Generic Function Match

classes. Modules are also often used just to group a set of related functions, like I/O routines. This section addresses the problem of locating modules in a component library.

Recall that whereas the signature of a function is simply its type, τ, the signature of a module is an interface, $\mathcal{I}$. A module's interface is a pair, $\langle \mathcal{I}_T, \mathcal{I}_F \rangle$, where $\mathcal{I}_T$ is a multiset of user-defined types and $\mathcal{I}_F$ is a multiset of function types.[3] For a library interface, $\mathcal{I}_L = \langle \mathcal{I}_{LT}, \mathcal{I}_{LF} \rangle$, to match a query interface, $\mathcal{I}_Q = \langle \mathcal{I}_{QT}, \mathcal{I}_{QF} \rangle$, there must be a correspondence between $\mathcal{I}_{LF}$ and $\mathcal{I}_{QF}$. This correspondence varies for exact and relaxed module match.

3.1 Exact Match

Definition 3.1.1 (*Exact Module Match*).

$$\begin{aligned} \textit{M-match}_E(\mathcal{I}_L, \mathcal{I}_Q) \;=\; & \exists \text{ a total mapping } U_F : \mathcal{I}_{QF} \rightarrow \mathcal{I}_{LF} \text{ such that} \\ & U_F \text{ is one-to-one and onto, and} \\ & \forall\ \tau_q \in \mathcal{I}_{QF}, \mathit{match}_E(U_F(\tau_q), \tau_q) \end{aligned}$$

U_F maps each query function type τ_q to a corresponding library function type, $U_F(\tau_q)$. Since U_F is one-to-one and onto, the number of functions in the two interfaces must be the same (i.e., $|\mathcal{I}_{LF}| = |\mathcal{I}_{QF}|$). The correspondence between each τ_q and $U_F(\tau_q)$ is that they satisfy the exact function match, $match_E$, defined in Section 2.1. That is, the types match modulo renaming of type variables and user-defined type operators.

We could additionally require a mapping between user-defined types ($\mathcal{I}_{LT}$ and $\mathcal{I}_{QT}$), but for the most part, matching function types suffices, since for τ_q and $U_F(\tau_q)$ to match, any user-defined types must match. So any user-defined type that appears in the domain or range of a function type in one interface must match

[3]For useful feedback to the user, we would need to associate names with the function types, but this is not necessary in the definition.

a user-defined type in the other interface. Not having a separate mapping precludes matching where one user-defined type in $\mathcal{I}_{QT}$ matches more than one user-defined type in $\mathcal{I}_{LT}$ (or vice versa). This case is not likely to occur in practice since programmers typically define only one user-defined type per module.

Query $Q1$, shown below, is the interface of a module that defines an abstract container type and four basic functions on the container:

$$\begin{array}{lll} \text{Q1:} & \mathcal{I}_{QT} = \{\ \alpha\ C\ \} \\ & \mathcal{I}_{QF} = \{\ \mathit{unit} \rightarrow \alpha\ C, \\ & \qquad\qquad (\alpha,\ \alpha\ C) \rightarrow \alpha\ C, \\ & \qquad\qquad \alpha\ C \rightarrow (\alpha,\ \alpha\ C), \\ & \qquad\qquad \alpha\ C \rightarrow \mathit{int}\ \} \end{array}$$

This matches the interface for the $QUEUE$ module in Figure 1 with the the obvious mapping from function types in $\mathcal{I}_{QF}$ to function types in $QUEUE$. Each of the exact function matches renames the user-defined type operator T to C.

Exact module match is rather restrictive. We define two forms of relaxed module match by (1) modifying the mapping U_F and (2) replacing the definition of function match, $match_E$.

3.2 Partial Match

Should a querier really have to specify all the functions provided in a module in order to find it? A more reasonable alternative is to allow the querier to specify a subset of the functions (namely, only those that are of interest) and match a module that is more *general* in the sense that it may contain functions in addition to those specified in the query.

Definition 3.2.1 (*Generalized Module Match*).

> $M\text{-}match_{gen}(\mathcal{I}_L, \mathcal{I}_Q)$ is the same as $M\text{-}match_E(\mathcal{I}_L, \mathcal{I}_Q)$ except U_F need not be onto.

Thus whereas with $M\text{-}match_E(\mathcal{I}_L, \mathcal{I}_Q)$, $|\mathcal{I}_{LF}| = |\mathcal{I}_{QF}|$, with $M\text{-}match_{gen}(\mathcal{I}_L, \mathcal{I}_Q)$, $|\mathcal{I}_{LF}| \geq |\mathcal{I}_{QF}|$, and $\mathcal{I}_{LF} \supseteq \mathcal{I}_{QF}$ under the appropriate renamings. A query like $Q1$ but without the function type $\alpha\ C \rightarrow \mathit{int}$ would match $QUEUE$ under the generalized module match definition.

Definition 3.2.2 (*Specialized Module Match*).

$$M\text{-}match_{spec}(\mathcal{I}_L, \mathcal{I}_Q) = M\text{-}match_{gen}(\mathcal{I}_Q, \mathcal{I}_L)$$

With specialized module match, a library need not have all the functions defined in the query. As with specialized and generalized match for functions, specialized module match is the converse of generalized module match.

3.3 Relax* Match (Using Relaxed Function Matches)

In the definition of exact module match we used the exact match predicate, $match_E$, to determine whether a function in the query interface matches one in the library interface. An obvious relaxation is to use a relaxed match on functions instead of exact match.

Definition 3.3.1 (*Relax* Match*).

$$\textit{M-match}_{relax^*}(\mathcal{I}_L, \mathcal{I}_Q, \mathcal{M}_F) = \exists \text{ a mapping } U_F : \mathcal{I}_{QF} \rightarrow \mathcal{I}_{LF} \text{ such that}$$
$$U_F \text{ is one-to-one and onto, and}$$
$$\forall\ \tau_q \in \mathcal{I}_{QF}, \mathcal{M}_F(U_F(\tau_q), \tau_q)$$

The only difference between relax* match and exact module match is that relax* match uses its parameter, $\mathcal{M}_F$, to match functions, instead of fixing function match to be exact, $match_E$. Thus, exact module match is trivially defined in terms of the above definition: $\textit{M-match}_E(\mathcal{I}_L, \mathcal{I}_Q) = \textit{M-match}_{relax^*}(\mathcal{I}_L, \mathcal{I}_Q, match_E)$. The match parameter ($\mathcal{M}_F$) gives us a great deal of flexibility, allowing any of the function matches defined in Section 2 to be used in matching the individual function types in a module interface.

What this definition makes clear in a concise and precise manner is the orthogonality between function match and module match.

3.4 Composition of Module Matches

As with function matches, we can compose module matches. Since specialized module match is defined in terms of generalized module match, we need only consider the composition of generalized module match and relax* match. The order of the composition does not matter, since generalized match affects the mapping U_F while relax* match changes only the function match used.

Definition 3.4.1 (*Generalized Relax* Match*).

$\textit{M-match}_{gen\text{-}relax^*}(\mathcal{I}_L, \mathcal{I}_Q, \mathcal{M}_F)$ is the same as $\textit{M-match}_{relax^*}(\mathcal{I}_L, \mathcal{I}_Q, \mathcal{M}_F)$ except U_F need not be onto.

We present this as a separate definition because we expect this combined relaxed match to be the most common use of module match in practice.

Query $Q2$ is another example of a module query. This query contains only two function types:

$$\begin{aligned} \text{Q2:}\quad \mathcal{I}_{QT} &= \{\ \alpha\ C\ \} \\ \mathcal{I}_{QF} &= \{\ \textit{unit} \rightarrow \alpha\ C, \\ &\qquad (\alpha\ C, \alpha) \rightarrow \alpha\ C\ \} \end{aligned}$$

Under generalized module match, this query would match only the *SET* module (with U_F mapping the query functions to *create* and *insert* (or *delete*)). Under generalized relax* match, using function reorder match, the query matches not only *SET* but also the *QUEUE* module (with U_F mapping the query functions to *create* and *enq* and reordering the input arguments to *enq*).

3.5 Generic Match

As with function matching, we define a generic form of module match, based on the relation between the function types of the query and library interfaces.

Definition 3.5.1 (*Generic Module Match*).

M: Library Interface, Query Interface → Boolean
$M(\mathcal{I}_L, \mathcal{I}_Q) = \mathcal{I}_{QF}\ R^{\mathcal{M}}\ \mathcal{I}_{LF}$

Match	$R^{\mathcal{M}}$	$\mathcal{M}$
Exact	$=$	$match_E$
Generalized	$\supseteq$	$match_E$
Specialized	$\subseteq$	$match_E$
Relax*	$=$	any
Generalized Relax*	$\supseteq$	any

Table II. Instantiations of Generic Module Match

The generic module match definition has two implicit parameters, R and $\mathcal{M}$. R is a set relation between the multisets $\mathcal{I}_{QF}$ and $\mathcal{I}_{LF}$. R is parameterized by the function match relation $\mathcal{M}$ which compares the individual elements of the set. Table II shows how R and $\mathcal{M}$ are instantiated for the module matches defined in this section. The set relation R is either $=$, $\subseteq$ or $\supseteq$. The function match $\mathcal{M}$ is either $match_E$ (exact function match) or "any", in which case the match can be any of the function matches defined in Section 2.

4. EXPERIENCE USING SIGNATURE MATCHING

4.1 Applications

Most applications of signature matching use a query to describe and retrieve a subset of components of interest from the library. These subsets can be used to locate components for reuse or to analyze, browse, or filter the library. We can also use the match definitions themselves to define a structure over the library for indexing. In this section we discuss these applications with examples taken from actual use of our function signature matcher.

The library we used contains 1451 SML functions, gathered from three locally available libraries: the Edinburgh library (688 functions) [Berry 1991], the SML/NJ library (362 functions) [ATT 1993], and a library of local contributions (401 functions) [Tarditi and Rollins 1993]. While our implementation (and thus examples) only addresses function signature matching, the applications hold equally well for module signature matching.

4.1.1 *Reuse.* The most obvious and widely discussed application of signature matching is to retrieve components for reuse. In order for signature matching to be successful for reuse, the matcher should be easy to use, 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), and the component should be relatively easy to use once found. The following examples are drawn from use of our signature matcher 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 matching to retrieve functions for reuse.

In the first example, 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 would take a boolean value b and an integer n and generate a list

of length n where each element has value b. Since it seemed likely that a function in a library would be more general about the type of the list, we used the query $(\alpha, int) \rightarrow \alpha\ list$ with relaxations reorder and uncurry. This search resulted in the function *create* (with type $int \rightarrow \alpha \rightarrow \alpha\ list$, from the *List* module in the Edinburgh library), which does exactly what we wanted. Interestingly, 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 generalize, we get 28 functions that match instead of just *create*, since both *bool* and *int* can be generalized.

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 (For example, convert ["Parser", "Table", "T"] to "Parser.Table.T"). We used the query $(string\ list, string) \rightarrow string$ with relaxations reorder and uncurry. 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. We performed this search and found the *implode* function, which does close to what we wanted but is not easily modifiable, and the function *firstLine*, 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). The related query $string\ list \rightarrow string$ with no relaxations results in six matches, two of which are the functions *pathImplode* and *implodePath* from the SML/NJ library and local library respectively. Both of these functions take a list of strings and return a string which is the concatenation of those strings with "/"s between the strings (to form a path name), but with very different implementations. Either of these functions is easy to modify to do what we want by replacing "/"s with "."s. This example shows that a relaxation to allow more or fewer arguments in a tuple would be useful.

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, and found the *zip* function in all three libraries. He was able to reuse *zip* as is in his program.

An example that gave us a surprising result came from performing the query $\alpha\ list \rightarrow \alpha$ using specialized match. We expected this query to return functions such as one that selects an element from a list (*hd*); we also retrieved the *pathImplode* and *implodePath* functions discussed above. This example illustrates the tradeoffs in using relaxations that instantiate type variables (generalize and specialize); increasing the chance of finding a potentially useful function (i.e., increasing recall) may reduce the precision of the query and match with more useless components.

An example where we really wished we had a signature matcher was in implementing in Lisp the gnu-emacs interface for our signature matcher, even though gnu-emacs has a good keyword search facility (`apropos`) and we had a reasonable manual. Lisp has over 500 built-in functions listed in the manual index [Steele Jr. 1984]. 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, or we could look up "write" 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 matching with the query $(string, file) \rightarrow unit$.

4.1.2 *Analysis.* Another use of signature matching is to analyze a software library. A colleague of ours wanted to gather statistics about what percentage of functions in the libraries are higher-order. He performed the query $\alpha \rightarrow \beta \rightarrow \gamma$ with the specialize relaxation and found 577 functions out of the 1451 in the library (40%). To find out which functions return real numbers, we used the query $\alpha \rightarrow real$ with the specialize relaxation, which returns 19 functions.

A question like "how many functions take a real number as one of the inputs?" is more difficult because a query like $(\alpha, real) \rightarrow \beta$ with relaxations specialize and reorder only matches functions with a two-element tuple input. However, we can answer the question with a small set of queries and a simple inspection of the results. The following table shows the results of a sequence of queries with relaxations specialize, reorder and uncurry:

Query	**Number of matches**
$real \rightarrow \alpha$	25
$(real, \alpha) \rightarrow \beta$	24
$(real, \alpha, \beta) \rightarrow \gamma$	1
$(real, \alpha, \beta, \gamma) \rightarrow \delta$	0

Once we analyze the results to delete the overlap between the first two queries for functions of the form $real \rightarrow \alpha \rightarrow \beta$, we have 40 functions that take a real number as part of the input. Can we make a more general statement about the number of elements in an input tuple? The following table shows the results of queries with 2, 3 and 4 element tuples using specialized match. Queries with 5, 6 or 7 element tuples do not find any additional matches.

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

4.1.3 *Browsing.* Signature matching is also useful if a user wants to become more familiar with the software library by browsing through the library. Perhaps the user is new to a particular programming language and its facilities; he or she can use signature queries to explore the library to learn about what is available and learn the "style" of the language or library. For example, in ML, user-defined types are often labeled as simply T, as one could quickly notice from a query like $(\alpha, \alpha\ X) \rightarrow \alpha\ X$ (recall that our signature matching allows renaming of user-defined types). Analyzing the results of our query about the percentage of higher-order functions discussed in the previous paragraph, we can observe the distinctions in styles even among ML libraries: 51% of the functions in the Edinburgh library are higher-order, compared with 27% of the SML/NJ library and 32% of the local library.

4.1.4 *Filtering.* We consider a signature matcher as one in a collection of tools for the software engineer, so we can naturally think about composing retrieval tools. In this case, a coarser-grained tool acts as a "filter" for a more fine-grained tool. For example, if we have, in addition to the signature matcher, a tool to match *specifications* of components (Section 6), we can use the signature matcher

to first check that components have matching signatures before invoking the (more expensive and slower) specification matcher.

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

(1) Let $L1$ be the result of the query $real \rightarrow \alpha$ on the full library.

(2) Let $L2$ be the result of the query $real \rightarrow \alpha \rightarrow \beta$ on $L1$.

(3) Let $L3 = L1 - L2$. $L3$ contains the functions of the desired form.

In our example library, $L1$ contains 25 functions, and $L2$ contains 10, leaving us with 15 functions of the desired form. Ten of those have type $real \rightarrow real$, four have type $real \rightarrow int$, and one has type $real \rightarrow (real, real)$.

4.1.5 *Indexing.* Another application of the match definitions is indexing a library. Indexing does not use the search facility of signature matching at all, but rather uses the relationships defined by the various match definitions. Generalized (or specialized) match defines a partial ordering on a library with the $\geq$ relation. Similarly the transformation relaxations (reorder, uncurry and their composition) define equivalence classes on the library's components. We can calculate these relations in a pre-processing step and use them to navigate through the library or even to store closely-related functions together to localize disk access. 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 our sample library in Figure 1, they might want to know if there is a more general sorting function that works for arbitrary types of list elements. If there were an signature match-based index for the library, they could simply ask to see all functions whose types are more general than *intsort*.

Indexing based on module signature matching produces a kind of subtype hierarchy. If we consider the functions of a module to correspond to methods, and let $\mathcal{I}_Q$ represent the supertype and $\mathcal{I}_L$ the subtype, then $M\text{-}match_{gen}(\mathcal{I}_Q, \mathcal{I}_L)$ corresponds to the subtype supplying all the methods with the same types as those of the supertype plus perhaps some more.

4.2 Implementation Details

Building a signature matcher for a strongly-typed programming language is not much more involved than modifying its type checker. We have integrated a signature matching facility into our local Standard ML (SML) programming environment. Our current implementation, itself written in SML, supports the function matches defined in this paper. The algorithms for generalized and specialized match are modifications of Robinson's unification algorithm, as presented by Milner [Milner 1978]. The algorithms for the other matches are straightforward; we use a simple transformation of the type for uncurry match, and a backtracking algorithm for reorder match. On a test suite of 10 queries on each of the 16 combinations

```
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

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

Fig. 2. Output Buffer of Signature Matcher

of relaxations using our library of 1451 functions, 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 relaxations.

Our user interface is intentionally simplistic – it is just gnu-emacs [Stallman 1986] and a mouse. The user defines the query and can select 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 shows the results of a query as they appear in the emacs buffer. Type notation is from SML: `*` is the tuple constructor, `->` is the function constructor, `'a, '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 matching from their normal software development environment. We wanted to make signature matching as easily available for use as string searching.

5. RELATED WORK

Closely related work on signature matching has focused on either signature matching as an application of a particular theoretical definition of type isomorphism or as a retrieval tool to be used in conjunction with other tools. Rittri [1989] defines an extension of $match_{reorder^*} \circ match_{uncurry^*}$ by identifying types that are isomorphic in a Cartesian closed category. He has also developed an algorithm to check for more general types modulo this isomorphism [Rittri 1992], similar to our $match_{reorder^*} \circ match_{uncurry^*} \circ match_{gen}$. He has implemented both matches. Di Cosmo [1992] extends Rittri's approach with a theory that also handles isomorphisms of types with *let* expressions. Runciman and Toyn [1989] assume that queries are constructed by example or by inference from context of use. They use queries to generate a set of keys, performing various operations on the set to permit more efficient search. The match they ultimately perform is similar to our unify match. Rollins and Wing [1991] implemented a system in Lambda-Prolog

that includes the equivalent of $match_{reorder^*} \circ match_{uncurry^*}$. The NORA software development environment includes a retrieval tool that uses a match similar to $match_{reorder^*} \circ match_{unify}$; they use variables in types to allow incomplete components [Snelting et al. 1991]. Stringer-Calvert [1994] has implemented a signature matcher for Ada packages. His match definitions are based on the ones we present here.

Our work is unique in three ways. First, we have identified a small set of primitive function matches that can be combined in useful ways. Definitions of signature matching given by others are special cases of our more general approach; we can succinctly characterize their definitions (as above) and do so in a common formal framework. We support orthogonality of concepts, allowing the user to "pick and choose" whichever match is desired, perhaps through a combination of more primitive matches. Second, all previous work has focused solely on matching at the function level. We extend signature matching to include matching on modules 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. Finally, we go beyond retrieval for reuse and present four additional applications of signature matching.

Additional less closely related work on matching software components divides into three categories based on the kind of information being matched: specifications, code or text. Specification matching [Zaremski and Wing 1995; Rollins and Wing 1991; Fischer et al. 1994; Perry 1989; Jeng and Cheng 1992; Mili et al. 1994; Katz et al. 1987; Yellin and Strom 1994] defines match in terms of formal specifications. Specification matching allows greater precision in matching than does signature matching, but requires specifications of components and a more expensive match. Another class of matches [Paul and Prakash 1994; Consens et al. 1992] allows 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. Other research 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 [Arnold and Stepoway 1987; Prieto-Díaz 1989], or in English, based on text from documentation of the components [Maarek et al. 1991]. In some cases, additional structural information can be added in semantic nets [Helm and Maarek 1991; Ostertag et al. 1992].

All of these other approaches require at the very least that the user learn another language (except for the natural language-based approach in Maarek et.al.[1991]). Except for Maarek et al. [1991] and Paul and Prakash [1994], 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. Specification and information retrieval approaches typically only address retrieval for reuse as an application; code-based approaches focus mostly on browsing or analyzing the actual code. We show how signature matching addresses all these applications and more.

As mentioned in the introduction, we view signature matching as a complementary approach to more traditional information retrieval techniques. A user can

choose 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 as we discussed in Section 4.1.

6. SUMMARY AND FUTURE WORK

This paper lays the foundation for the use of signature matching as a practical tool for using software libraries. We present precise definitions of a variety of matches at both the function and module levels. Areas for further work include extending signature matching to other languages and going beyond signatures to specification matching.

The basic ideas and definitions of signature matching apply to any strongly-typed programming language, but many of the relaxed matches depend on specific features of the language. Generalized and specialized match only apply to languages with polymorphic types, uncurry match to those that permit higher-order functions. There may be additional relaxations that would be useful in other languages. For example, a language may distinguish between mutable and immutable parameters. If there are two functions that perform the same operation but one does so by creating a new object $((\alpha, \alpha\ T) \rightarrow \alpha\ T)$ and the other by mutating an input object $((\alpha, \alpha\ T) \rightarrow \text{unit})$ we might like to be able to say these are "the same" under some relaxation. We also have not elaborated relaxed matching by taking into consideration exceptional return values.

In the introduction we observed that signature matching is an instance of using domain-specific information to do search. In an ideal software library, domain-specific information would include not just signature information, but *formal specifications* of the behavior of each component. Given such specifications, e.g., pre-/post- conditions for functions, we could then add to our arsenal of software library tools a *specification matcher*, using specifications, not just signatures, to match components. Consider the query $(\alpha\ T, \alpha\ T) \rightarrow \alpha\ T$ which matches the *union*, *intersection* and *difference* functions on sets. Specification matching would let us distinguish among these three since their *behaviors* differ even though their types are the same. We are pursuing specification matching in the context of Larch [Guttag and Horning 1993] and Larch/ML [Wing et al. 1993] at Carnegie Mellon: we have built a prototype specification matcher using the Larch Prover[Garland and Guttag 1991] as the basis of our match engine. Unfortunately we cannot as yet expect programmers to document their program components with formal specifications.[4] Signature matching backs off from this more ambitious approach.

Signature matching takes advantage of information about program modules that is essentially free. Function types and module interfaces are either generated automatically by type inference systems, or programmers must provide them for the compiler anyway. Implementing signature matching requires nothing more sophisticated than unification, a standard algorithm already used in some compilers to do type inference. In return we get a useful tool for organizing, exploring and retrieving from software libraries.

[4] Though, if we provided them with efficient specification matchers, maybe there would be additional incentive to write formal specifications—a chicken-and-egg problem!

ACKNOWLEDGMENTS

We thank Gene Rollins for his help in modifying the SML compiler to provide the necessary hooks for the signature matcher, Gene Rollins and Chris Okasaki for testing the signature matcher and suggesting improvements, Scott Nettles for asking for statistics on higher-order functions, Roberto Di Cosmo for clarifying some of the relations between our work, and the editor and referees for their valuable comments and pointers to additional related work.

REFERENCES

Agresti, W. W. and McGarry, F. E. 1987. The Minnowbrook workshop on software reuse: A summary report. In W. Tracz (Ed.), *Tutorial: Software Reuse: Emerging Technology*, pp. 33–40. IEEE Computer Society Press.

Arnold, S. P. and Stepoway, S. L. 1987. The REUSE system: Cataloging and retrieval of reusable software. In W. Tracz (Ed.), *Tutorial: Software Reuse: Emerging Technology*, pp. 138–141. IEEE Computer Society Press.

ATT 1993. The Standard ML of New Jersey library reference manual. Technical report (Feb.), AT&T Bell Laboratories.

Berry, D. 1991. The Edinburgh SML library. Technical Report ECS-LFCS-91-148 (April), University of Edinburgh.

Biggerstaff, T. J. and Perlis, A. J. (Eds.) 1989. *Software Reusability Vol. 1: Concepts and Models*. ACM Press, N.Y.

Bischofberger, W. R. 1992. Sniff – a pragmatic approach to a C++ programming environment. In *USENIX C++ Conference*, pp. 67–81.

Consens, M., Mendelzon, A., and Ryman, A. 1992. Visualizing and querying software structures. In *Proceedings of the* 14^{th} *International Conference on Software Engineering*, pp. 138–156. IEEE Computer Society Press.

Di Cosmo, R. 1992. Type isomorphisms in a type-assignment framework. In *The* 19^{th} *Annual Symposium on Principles of Programming Languages*, pp. 200–210.

Field, A. J. and Harrison, P. G. 1988. *Functional Programming*. Addison-Wesley.

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

Fontana, M. and Neath, M. 1991. Checked out and long overdue: Experiences in the design of a C++ class library. In *USENIX C++ Conference*, pp. 179–191.

Garland, S. J. and Guttag, J. V. 1991. A guide to LP, the Larch Prover. Report 82 (December), DEC Systems Research Center, Palo Alto, CA.

Guttag, J. and Horning, J. 1993. *Larch: Languages and Tools for Formal Specification*. Springer Verlag.

Helm, R. and Maarek, Y. S. 1991. Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries. In *OOPSLA Conference Proceedings*, pp. 47–61.

IEEE 1984. IEEE Transactions on Software Engineering. SE-10(5).

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

Katz, S., Richter, C. A., and The, K.-S. 1987. PARIS: A system for reusing partially interpreted schemas. In *Proceedings of the* 9^{th} *International Conference on Software Engineering*, pp. 377–385. IEEE Computer Society Press.

Maarek, Y. S., Berry, D. M., and Kaiser, G. E. 1991. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering 8,* 17 (Aug.), 800–813.

Mili, A., Mili, R., and Mittermeir, R. 1994. Storing and retrieving software components: A refinement-based approach. In *Proceedings of the* 16^{th} *International Conference on Software Engineering*. IEEE Computer Society Press.

Milner, R. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17,* 3 (Dec.), 348–375.

Milner, R., Tofte, M., and Harper, R. 1990. *The Definition of Standard ML.* MIT Press.

Ostertag, E., Hendler, J., Prieto-Díaz, R., and Braun, C. 1992. Computing similarity in a reuse library system: An AI-based approach. *ACM Transactions on Software Engineering and Methodology 1,* 3 (July), 205–228.

Paul, S. and Prakash, A. 1994. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering 6,* 20 (June), 463–475.

Perry, D. E. 1989. The Inscape environment. In *Proceedings of the* 11^{th} *International Conference on Software Engineering*, pp. 2–12. IEEE Computer Society Press.

Prieto-Díaz, R. 1989. Classification of reusable modules. In T. J. Biggerstaff and A. J. Perlis (Eds.), *Software Reusability Vol. 1: Concepts and Models*, pp. 99–123. N.Y.: ACM Press.

Rittri, M. 1989. Using types as search keys in function libraries. *Conference on Functional Programming Languages and Computer Architectures*, 174–183.

Rittri, M. 1990 (reprinted with corrections May 1992). Retrieving library identifiers via equational matching of types. Technical Report 65, Programming Methodology Group (Jan.), Department of Computer Sciences, Chalmers University of Technology and University of Göteborg.

Rollins, E. J. and Wing, J. M. 1991. Specifications as search keys for software libraries. In *Proceedings of the Eighth International Conference on Logic Programming.* MIT Press.

Runciman, C. and Toyn, I. 1989. Retrieving re-usable software components by polymorphic type. *Conference on Functional Programming Languages and Computer Architectures*, 166–173.

Snelting, G., Grosch, F.-J., and Schroeder, U. 1991. Inference-based support for programming in the large. In A. van Lamsweerde and A. Fugetta (Eds.), 3^{rd} *European Software Engineering Conference*, Number 550 in Lecture Notes in Computer Science, pp. 396–408. Springer Verlag.

Stallman, R. 1986. *GNU Emacs Manual.* Free Software Foundation, Combridge, Mass.

Steele Jr., G. L. 1984. *Common Lisp, The Language.* Digital Press.

Stringer-Calvert, D. 1994. Signature matching for Ada software reuse. Master's thesis, University of York.

Tarditi, D. and Rollins, G. 1993. Local guide to Standard ML. Technical report (March), CMU.

Tesler, L. 1981. The Smalltalk environment. *BYTE 6,* 8 (Aug.), 90–147.

Wing, J., Rollins, E., and Zaremski, A. M. 1993. Thoughts on a Larch/ML and a new application for LP. In U. Martin and J. M. Wing (Eds.), *First International Workshop on Larch.* Springer Verlag.

Yellin, D. M. and Strom, R. E. 1994. Interfaces, protocols, and the semi-automatic construction of software adaptors. *OOPSLA Conference Proceedings, ACM SIGPLAN Notices 29,* 10 (Oct.), 176–190.

Zaremski, A. M. and Wing, J. M. 1995. Specification Matching of Software Components. In *Third ACM SIGSOFT Symposium on the Foundations of Software Engineering.* Also published as CMU Tech. Report CS-95-127.

Received August 1994; revised March 1995; accepted April 1995