Signature Matching: A Key to Reuse

Amy Moormann Zaremski and Jeannette M. Wing
(amy@cs.cmu.edu and wing@cs.cmu.edu)
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213

Abstract

Software reuse is only effective if it is easier to locate (and appropriately modify) a reusable component than to write it from scratch. We present *signature matching* as a method for achieving this goal by using signature information easily derived from the component. We consider two kinds of software 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 briefly describe an experimental facility written in Standard ML for performing signature matching over a library of ML functions.

1 What is Signature Matching?

Software reuse sounds like a good idea. It promises advantages like reducing the time and cost spent on programming, increasing programmers' productivity, and increasing program reliability [BP89, AM87, IEE84, Pre87]. But why doesn't it work in practice? One reason is that it is hard to find things. As libraries of software components get larger, this problem will get worse. Reuse is only worth it if it is easier to locate (and appropriately modify) a reusable component than to write it from scratch.

Today, if we want to find some desired component, we could use the component's name—if we are lucky enough to know, remember, or guess it. We could browse through the library itself, or perhaps an index into the library (for example, as with a Smalltalk browser). Given that the components over which we are searching are program units (e.g., Pascal procedures, C functions, Ada packages, C++ or Smalltalk classes, or Modula-3 or ML modules), then we have another means for retrieval: *signature matching.* This paper presents the foundations for what signature matching means and briefly describes a signature matching facility we have built and integrated into our local ML programming environment.

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`.[1]

```
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,
```

Figure 1: Three ML (Signature) Modules

If we are looking for a specific function, rather than perform a query based on its name, we could perform a query based on 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, which takes a list of

[1]ML *signature* modules are akin to Ada *definition* modules and Modula-3 *interface* modules, ML implementations are written in modules called *structures* [MTH90]

objects of some type, α, and returns an object of that type. If we are looking for a module, we could perform a query based on its *interface*, which is a multiset of user-defined types and a multiset of function types. For example, SET has one user-defined type, $\alpha\ T$, and seven function types. 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.

Signature matching is the process of determining when a library component "matches" a query. 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. 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 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. *Signature Match*: Query Signature, Match Predicate, Component Library $\rightarrow$ Set of Components

$$\textit{Signature Match}(q, M, C) = \{c \in C : M(c, q)\}$$

In other words, given a 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* (type) *match* and *module* (interface) *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).

In a broader context, signature matching can be viewed as another 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.

By not requiring users to know the name (or unique identifier) of what they are searching for, we can also view signature matching as an example of content-addressable search. For example, users formulate queries in terms of key-value pairs to retrieve records from a relational database. Hence, the intended pun in our title: signature matching is a "key" to software reuse.

Signature matching is useful not only to retrieve components. Software developers might use a signature matcher to filter out the bulk of library components from further consideration. They can also use signature matchers to browse a software library in a structured way, e.g., by exploiting the partial ordering induced by function types. 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 to use as doing a search on a string pattern.

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 describe our signature matching facility which we have used in searching over a collection of Standard ML modules. 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[FH88]. A *type* is either a type *variable* ($\in$ *TypeVar*, denoted by Greek letters) or a type *operator* applied to other types. Type operators (*TypeOp*) 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.[2] 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* ($\tau = \tau'$) if either they are the same type variable, or $\tau = typeOp(\tau_1, ..., \tau_n)$, $\tau' = typeOp'(\tau'_1, ..., \tau'_n)$, $typeOp = typeOp'$, and $\forall\ 1 \leq i \leq n, \tau_i = \tau'_i$. *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:

$$[\beta/\gamma][\alpha/\beta](\beta \rightarrow \gamma) = [\beta/\gamma](\alpha \rightarrow \gamma) = (\alpha \rightarrow \beta)$$

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 type operator names as variables for the purposes of renaming, since different users may use a different name for the same type

[2] 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. Renaming sequences may include both type variable renaming and user-defined type operator renaming:

$$[\beta/\alpha][C/T](\alpha\ T \to \alpha) = (\beta\ C \to \beta)$$

We will use V for a sequence of variable renamings and U for a sequence of more general substitutions. We distinguish between variable renaming and variable substitution since we allow variable renaming in exact match and variable substitution in some relaxed matches.

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 match*, $M(\tau_l, \tau_q)$, as follows:

Definition. (*Generic Function Match*)
M: Library Type, Query Type $\to$ Boolean

$$M(\tau_l, \tau_q) = T_l(\tau_l)\ R\ T_q(\tau_q)$$

where T_l and T_q are transformations (e.g., reordering) and R is some relationship between types (e.g., equality). 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 ($=$) 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*, the general signature match function defined in Section 1.

2.1 Exact Match

Definition: (*Exact Match*)

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

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 $match_E(\tau_l, \tau_q) == (\tau_l = \tau_q)$ 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 $V_1\tau_1 = V_2\tau_2$, we could construct a V' such that $V'\tau_1 = \tau_2$. (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, $\tau_l = (\alpha, \alpha) \to bool$ matches $\tau_q = (\beta, \beta) \to bool$ with the substitution $V = [\beta/\alpha]$. But $\tau_l = \alpha \to \beta$ and $\tau_q = \gamma \to \gamma$ do not match because once we substitute γ for α to get $\gamma \to \beta$, 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 $\tau_q = (\alpha \to \gamma) \to \alpha\ list \to \gamma\ list$ 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 $\tau_q = (\alpha, \alpha\ C) \to \alpha\ C$. 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

As we have just seen, exact match is a useful starting point, but may miss useful functions whose types are close but do not exactly match the query. Exact match requires a querier 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* $\to$ *int list*, could just as easily use an instantiation of a more general function, e.g., α *list* $\to$ α *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 querier 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 querier. We define *generalized* and *specialized* match to address both of these cases.

Referring back to our definition 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, $\alpha \to \alpha$ is a generalization of infinitely many types, including $int \to int$ and $(int, \beta) \to (int, \beta)$, 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: (*Generalized Match*)

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

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 substitutions are variables, 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

$$\tau_q = (int \to bool) \to int\ list \to bool\ list$$

This query does not match exactly with any function in our library. But a generalized match would return map for this

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

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

Figure 2: Definitions of *uncurry* and *uncurry**

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: (*Specialized Match*)

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

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.

Note that 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 (alternate): (*Generalized, Specialized Match*)

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

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

We can even define $match_{gen}(\tau_l, \tau_q)$ as $U \tau_l = \tau_q$; 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 $match_E$ definition when we define the composition of different kinds of relaxed matches (Section 2.4).

Finally, using these alternate definitions of generalized and specialized match, we can define *type unification* [FH88] by combining these two relaxed matches and allowing the renaming to occur on either type.

Definition: (*Unify Match*)

$match_{unify}(\tau_l, \tau_q) =$
 $\exists$ a sequence of variable substitutions, U, 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* the order or form of parts of 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*, which produces an uncurried version of a given type, is defined in Figure 2. The uncurry transformation is non-recursive; any nested functions will not be uncurried. Figure 2 also defines a recursive version, *uncurry**. For example:

If
 $\tau = int \rightarrow int \rightarrow (int \rightarrow int \rightarrow bool) \rightarrow bool$
then
 $uncurry(\tau) = (int, int, (int \rightarrow int \rightarrow bool)) \rightarrow bool$
and
 $uncurry^*(\tau) = (int, int, ((int, int) \rightarrow bool)) \rightarrow bool$

We could also define a form of uncurry where the level of uncurrying is defined by the user: $uncurry^n(\tau)$ will uncurry n levels of curried functions.

Definition: (*Uncurry Match, Recursive Uncurry Match*)

$$\begin{aligned}&match_{uncurry}(\tau_l, \tau_q) = \\ &\quad match_E(uncurry(\tau_l), uncurry(\tau_q)) \\ &match_{uncurry^*}(\tau_l, \tau_q) = \\ &\quad match_E(uncurry^*(\tau_l), uncurry^*(\tau_q))\end{aligned}$$

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 *uncurry* (or *uncurry**) 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). For example, suppose a user needs a function to add an element to a collection. The query $\tau_q = \alpha \rightarrow \alpha\ T \rightarrow \alpha\ T$ does not match exactly with any functions in our library, but uncurry match would return the function enq on a queue.

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

One common use of tuples is to group multiple arguments to a function where 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 *permutation* σ is a one-to-one mapping with domain and range $1 \ldots n-1$ such that $\sigma(\tau) = (\tau_{\sigma(1)}, \ldots, \tau_{\sigma(n-1)}) \rightarrow \tau_n$.

Definition: (*Reorder Match*)

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

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. For this match to succeed, both τ_l and τ_q must be function types whose first arguments are tuples. Although we choose to apply the permutation transformation, σ, to the library type τ_l, we could equivalently apply the inverse, σ^{-1}, to the query type τ_q: $match_E(\sigma(\tau_l), \tau_q) = match_E(\tau_l, \sigma^{-1}(\tau_q))$.

Suppose we again are looking for a function that adds an element to a collection. To find it, we might pose the query, $\tau_q = (\alpha, \alpha\ T) \rightarrow \alpha\ T$. With exact match we would find the enq function on queues, but with reorder match we would additionally find the insert and delete functions on sets. The functions enq and insert are both potentially useful.

Two variations on reorder match are (1) to allow recursive permutations so that a tuple's component types may be reordered ($match_{reorder^*}$ and $match_{reorder^n}$); and (2) to allow 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. For example, in searching for a function to add to a collection, uncurry match on the query $\alpha \rightarrow \alpha\ T \rightarrow \alpha\ T$ retrieves enq but misses insert. To retrieve both functions with this query, we need a way to combine the different relaxed matches.

We deliberately gave our definitions in a form so that we can easily compose them. Each of the relaxed match definitions presented in Sections 2.2 and 2.3 (using the alternate definition of $match_{gen}$ and $match_{spec}$) can be cast in the general form:

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

A transformation pair, T, is a tuple (T_l, T_q) of transformations. T_l is a transformation on the library type, and T_q is a transformation on the query type. For $match_{uncurry}$, the "$\exists$" is not necessary, since there is only one possible uncurry transformation. For $match_{gen}$ and $match_{reorder}$, T_q is the identity function, and in $match_{spec}$, T_l is the identity function.

The *match composition* of two relaxed matches, denoted as $(match_{T1} \circ match_{T2})$, is defined by applying the inner (T2) transformation first:

Definition: (*Match Composition*).

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

Thus 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 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 $match_{gen}$,[3] we also exclude them in our analysis. Thus, there are three "basic" relaxed matches: $match_{gen}$, $match_{uncurry}$, and $match_{reorder}$. We now consider some of the interesting combinations of these relaxed matches, those we expect queriers to find useful. 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).

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

  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.

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

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

  τ_l and τ_q match if the uncurried form of τ_l is more general than the uncurried form of τ_q.

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

  τ_l and τ_q match if some permutation of τ_l is more general than τ_q.

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

  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 ($\tau_l = (int, int \rightarrow bool) \rightarrow intlist \rightarrow intlist$).

  $Match_{gen}$ 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. For example, suppose $\tau_q = (bool, int) \rightarrow (int, bool)$ 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 into type expressions that contain a tuple.

3 Module Matching

Function matching addresses the problem of locating a particular function in a component library. However, a programmer often needs a collection of functions, e.g,. one that provides a set of operations on an abstract data type. Most modern programming language explicitly support the definition of abstract data types through a separate modules facility, e.g., CLU clusters, Ada packages, or C++ 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.[4] 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 correspondences both between $\mathcal{I}_{LT}$ and $\mathcal{I}_{QT}$ and between $\mathcal{I}_{LF}$ and $\mathcal{I}_{QF}$. These correspondences vary for exact and relaxed module match.

3.1 Exact Match

Definition: (*Exact Module Match*)

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

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

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, but for the most part, matching function type matches 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 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.

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

Figure 3: A module query

Figure 3 is an example query that describes a module containing the definition of an abstract container type and a set of basic functions on the container. 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 in the above definition 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 the module? 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: (*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 that in Figure 3 but without the function type $\alpha\ C \rightarrow int$ would match QUEUE under the generalized module match definition.

Definition: (*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: (*Relax* Match*)

$$\begin{aligned} &M\text{-}match_{relax*}(\mathcal{I}_L, \mathcal{I}_Q, \mathcal{M}_F) = \\ &\quad \exists \text{ a mapping } U_F : \mathcal{I}_{QF} \rightarrow \mathcal{I}_{LF} \text{ such that} \\ &\quad\; U_F \text{ is one-to-one and onto, and} \\ &\quad\; \forall\ \tau_q \in \mathcal{I}_{QF}, \mathcal{M}_F(U_F(\tau_q), \tau_q) \end{aligned}$$

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: $M\text{-}match_E(\mathcal{I}_L, \mathcal{I}_Q) = M\text{-}match_{relax*}(\mathcal{I}_L, \mathcal{I}_Q, match_E)$. 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 can be 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: (*Generalized Relax* Match*)

$M\text{-}match_{gen-relax*}(\mathcal{I}_L, \mathcal{I}_Q, \mathcal{M}_F)$ is the same as $M\text{-}match_{relax*}(\mathcal{I}_L, \mathcal{I}_Q, \mathcal{M}_F)$ 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.

Figure 4 shows another example of a module query. This query contains only two function types. Under generalized module match, this query would match only the QUEUE module (with U_F mapping the query functions to `create` and `enq`). Under generalized relax* match, using function reorder match, the query matches not only QUEUE but also the SET module (with U_F mapping the query functions to `create` and `insert` (or `delete`), and reordering the input arguments to `insert`).

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

Figure 4: Another module query

4 An Experimental Signature Matching Facility

We have integrated a signature matching facility into our local Standard ML (SML) programming environment. Our current implementation, itself written in SML, supports a subset of 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 [Mil78]. The algorithms for the other matches are straightforward, but in some cases naive.

We used our facility to perform queries over a library of SML code consisting of 41 modules containing 245 functions. The majority of queries tested take less than .016 seconds (16 milliseconds) to complete. Figure 5 shows some actual output from an SML command-line-based version of the implementation. Type notation is from SML: `*` is the tuple constructor, `->` is the function constructor, `'a`, `'b` denote type variables (`''a` is notation for equality types, and `'1a` notation for reference types; we do not distinguish equality or reference types in the current implementation). Using exact match, the query, `('a * 'a T) -> 'a T`, returns two matches: `adjoin` from a Set implementation, and `cons` from a lazy stream implementation. Using uncurry match on the same query yields an additional match: the `enq` function on a sortable queue. The third query, `'a list -> 'b list -> ('a * 'b) list`, is drawn from a real use of our system by a coworker. He needed a function to take two lists and create a list of pairs of elements from those lists. The `zip` function that was found by this query did exactly what he wanted; moreover, he was able to use `zip`'s code in his program without any modification. The final query, `'a list -> 'a`, with specialized match, returns two functions. The first is the **hd** function, which removes an element from a list. The second function is **`implodePath`**, which takes a list of strings that represent a file path name (e.g., [“usr”, “amy”, “tex”]), and returns a string of the entire path name (“usr/amy/tex”). We were slightly surprised by the match with **`implodePath`**, exposing our hidden assumption that the functions matched by this query would return an element of the list (as **hd** does) rather than performing an operation to combine all elements of the list (as **`implodePath`** does).

Our user interface is simplistic: just gnu-emacs [Sta86] and a mouse. The user pre-loads a specified component library. The emacs command is similar to that for string searches. The result of a query is a list of functions whose types each matches the query, along with the pathname for the file that contains the function. 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. Thus we achieve the goal of having signature matching as easily available for use as string searching.

```
>> Query = ('a * 'a T) -> 'a T, matcher = exact
adjoin :((''a * ''a T) -> ''a T)        [11 test/Set.sml]
cons :(('1a * '1a T) -> '1a T)          [16 test/lstream.sml]
[CPU Time: 0.015625 secs., Elapsed Time: 0 secs.] (Objects Found: 2)
>> Query = ('a * 'a T) -> 'a T, matcher = uncurry
adjoin :((''a * ''a T) -> ''a T)         [11 test/Set.sml]
enq :('a -> ('a T -> 'a T))              [15 test/SortableQueue.sml]
cons :(('1a * '1a T) -> '1a T)           [16 test/lstream.sml]
[CPU Time: 0.0 secs., Elapsed Time: 0 secs.] (Objects Found: 3)
>> Query = 'a list  -> 'b list -> ('a * 'b) list, matcher = exact
zip :('a list -> ('b list -> ('a * 'b) list))  [11 test/ListFns.sml]
[CPU Time: 0.015625 secs., Elapsed Time: 0 secs.] (Objects Found: 1)
>> Query = 'a list -> 'a, matcher = specialize
hd :('a list -> 'a)                      [17 test/List.sml]
[[Substitute 'a/1 for 'a/~1]]
implodePath :(string list -> string)     [102 test/Pathname.sml]
[[Substitute string for 'a/~1]]
[CPU Time: 0.015625 secs., Elapsed Time: 0 secs.] (Objects Found: 2)
```

Figure 5: Sample Output from Function Matching

5 Related Work

Closely related work on signature matching has been done by Rittri, Runciman and Toyn, and Rollins and Wing. We review and compare our work with theirs below.

Mikael Rittri defines the equivalent of $match_{reorder^*} \circ match_{uncurry^*}$ by identifying types that are isomorphic in a Cartesian closed category [Rit89]. He has also developed an algorithm to check for more general types modulo this isomorphism, the equivalent of our $match_{reorder^*} \circ match_{uncurry^*} \circ match_{gen}$ [Rit92]. He has implemented both matches.

Runciman and Toyn assume that queries are constructed by example or by inference from context of use [RT89]. 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 also implemented a system in Lambda-Prolog that includes the equivalent of $match_{reorder^*} \circ match_{uncurry^*}$ [RW91]. They also extended signature matching to perform a restricted kind of *specification matching* (Section 6).

Our work is unique in two 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 just 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.

Less closely related work, but relevant to our context of software library retrieval, divides into two categories: text-based search and classification schemes. In text-based search, textual information, such as function names and comments, is used to locate desired software components. Attempts to make this approach more formal involve imposing a particular structure on comments to increase the accuracy of search. One example of such a system is REUSE [AS86]. Information about a component includes the component's name, author, and language processor to use.

Finally, some work on classifying software enables control over the search space. A classification scheme primarily facilitates browsing, as the goal is to provide a structure through which a user can navigate to locate a desired module. Prieto-Díaz [PD89] describes a method of classifying software using *facets* (e.g., the function the software performs, the objects manipulated by the software, the medium used, the type of system, the functional area, and the setting of the application). The REUSE system also includes a classification structure which is used to guide search through a menu system.

As mentioned in the introduction, we view signature matching as a complementary approach to these more traditional information retrieval techniques. For example, a classification scheme could be used in conjunction with signature matching for a "pipelined" query: The first stage of the pipeline would use a classification scheme to prune the search space for the second stage, which would use signature matching.

6 Summary and Future Work

This paper lays the foundation for the use of signature matching as a practical tool for the software engineer to aid in the retrieval of software for reuse. We present precise definitions for a variety of matches at both the function and module levels. Areas for further work include evaluating the usefulness of signature matching, defining additional relaxations, and going beyond signatures to specification matching.

We plan to do more extensive evaluation using our signature matching facility by conducting experiments with real users from our local SML research community (which includes over 20 graduate students, staff, and faculty). This evaluation will serve two purposes: to identify places for performance improvements, and more importantly, to provide feedback on the utility of signature matching for software reuse.

The existing set of relaxed function matches may still miss some potentially useful functions. To capture some of these additional matches, we will need to expand our type system to model additional characteristics of functions. Two examples of such characteristics are mutability and exceptional behavior. Even in "mostly" functional languages like ML, there may be functions that mutate objects. Thus 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 would like to be able to say these are "the same" under some relaxation. Similarly, we would like to be able to match functions that are the same except in their behavior under exceptional conditions.

In the introduction we argued 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 search tools a *specification matcher*, using specifications, not just signatures, as search keys. 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 [GH93] and Larch/ML [WRZ93] at Carnegie Mellon; Stringer-Calvert has proposed to do specification matching for Ada at the University of York [SC93]. Unfortunately we cannot as yet expect programmers to document their program components with formal specifications.[5] Signature matching backs off from this more ambitious approach.

Hence, signature matching offers the greatest amount of information about program modules for the least overhead. We can exploit information that programmers already must generate, i.e., function types and module interfaces, for the compiler. (Thus we get the search keys for free.) 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 retrieving software modules.

Acknowledgments

We thank Gene Rollins for his help in modifying the SML compiler so we could painlessly build and integrate the signature matcher. We also thank the referees of this paper for their helpful comments.

References

[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.

[AS86] Susan P. Arnold and Stephen L. Stepoway. The REUSE system: Cataloging and retrieval of reusable software. Technical Report 86-CSE-22, Southern Methodist University, October 1986.

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

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

[GH93] John Guttag and James Horning. *Larch: Languages and Tools for Formal Specification.* Springer Verlag, 1993.

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

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[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.

[Pre87] Roger S. Pressman. *Software Engineering: A Practioner's Approach.* McGraw-Hill, 2nd edition, 1987.

[Rit89] Mikael Rittri. Using types as search keys in function libraries. *Conference on Functional Programming Languages and Computer Architectures*, pages 174–183, September 1989.

[Rit92] 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, January 1990 (reprinted with corrections 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.

[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.

[SC93] David W.J. Stringer-Calvert. Reuse in Ada–finding components by specification. Proposal submitted for the degree of Master of Engineering, 1993.

[Sta86] Richard Stallman. *GNU Emacs Manual*, 1986.

[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.

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