Specifications as Search Keys for Software Libraries

Eugene J. Rollins and Jeannette M. Wing
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA 15213-3890

Contact: Gene Rollins, **Telephone:** (412) 268-5684
FAX: (412) 681-5739, **EMail:** rollins@cs.cmu.edu

October 3, 1990

1. Context and Motivation

Searching through a large repository of objects can be a tedious activity if a user cannot easily identify the object of interest. If the object is a file stored in a campus-wide distributed file system, the user might need to know some part of its name, e.g., a partial pathname, and possibly its location, e.g., the host name of the file server. If the object is a database record, the user might need to know the record's attribute names in order to formulate a query based on the attributes' values. What if the object is a component in a program module library?

We propose searching through software libraries using *specification matching*. We assume a specification, s_i, is associated with each component, p_i, in a software library of i components. For example, a procedure's specification might consist of a name, a signature, and a pair of pre- and post-conditions describing the procedure's behavior. Specification matching is the process of determining whether for a given query, q, and specification, s_i, s_i *satisfies* q. For example, if the query and specification languages are both drawn from the same logical language, then *satisfies* is logical implication; specification matching is the process of showing an implication holds.

The expressiveness of the query and specification languages will determine whether specification matching is decidable. At one extreme, as in traditional program verification, it might require some "heavy-duty" theorem proving with key insights provided by a human user. Through appropriate restrictions on the language and the meaning of *satisfies*, however, specification matching can be made practical and useful with or without any human guidance. This paper describes one instance of the more general idea of specification matching.

We use λProlog (pronounced "Lambda Prolog") [12] as our specification and query languages, and λProlog's built-in higher-order unification to do specification matching. The most significant advantage we gain in our use of λProlog is that unification gives us "for free" the desired theorem-proving power required to do specification matching in general.

Criteria for Effective Matching

Let R be the set of relevant objects (objects of interest to the user) in a library and Q be the set of objects resulting from a query. With an *ideal* match $Q = R$. We use *precision* and *recall* [18] as two measurements to determine how good a match is and define them as these ratios: $Precision = |R \cap Q| : |Q|$ and $Recall = |R \cap Q| : |R|$. Precision is the ratio between the number of relevant items returned and the number of all items returned. High precision means that a high percentage of the items resulting from a query are of interest to the user. Recall is the ratio between the number of relevant items returned and the number of all relevant items in the library. High recall means that few items of interest to the user are missed.

Scenario: A Pipelined Query

Imagine a user poses the query "What procedures in the library sort lists of elements of type α?" Breaking this single query into stages, the user might first retrieve all procedures whose signatures match α *list* $\rightarrow$ α *list*. At a second stage, the user might retain only those whose pre-condition states that the input list is non-empty and post-condition states that the output list is a permutation of the elements in the input list where the elements are arranged in ascending order. At a third stage, the user might retain only those procedures for which a certain property P holds; in our example, P might be the property that the procedure does not modify any of the elements in its list argument.

To test out our ideas, we built a prototype utility for specification matching in λProlog for a library of Standard ML (ML) functions [13], each of which we specify following Larch's two-tiered approach [7]. In Section 2 we highlight the features of λProlog, ML, and Larch necessary to understand our examples. We refer the reader to the references for details. Our prototype's basic utility supports signature matching, described in Section 3, as what might be done in the first stage of a pipelined query. Our extended utility supports more general specification matching, described in Section 4, and invokes the signature matching predicate. As we develop these matching predicates we explain how our changes affect the precision and recall of queries. Section 5 motivates our use of higher-order unification to do specification matching easily, but also notes where λProlog falls short of being an "ideal" specification and query language. Section 6 discusses related work. We close in Section 7 with a summary of our contributions; their potential influence in areas like formal specifications, software reuse and databases; and ideas for future work.

2. Highlights of λProlog, ML, and Larch

λProlog is a higher-order logic programming language; it treats functions and predicates as first-class objects. Its semantics is based on a logic that uses the mechanism of the typed λ-calculus for constructing predicate and function terms and permits quantification over such constructions. The first-order subset of λProlog is a typed dialect of Prolog. Standard ML [13] is a typed functional programming language that treats functions as first-class objects and supports user-defined abstract data types as well as a host of other modern programming language features. We specify each ML function or abstract type following Larch's two-tiered approach [7]. We defer a description of this approach to Section 4. For readers familiar with Larch, we essentially use λProlog as the assertion language of a Larch interface specification instead of the Larch Shared Language (a fragment of first-order logic with equality). We assume familiarity with Prolog and explain further details of λProlog, ML, and Larch as necessary.

3. Matching Signatures

Our basic utility for searching ML function libraries consists of λProlog clauses that specify ML types and function signatures (Section 3.1), and define signature matching (Sections 3.2 and 3.3).

3.1. Specifying ML Types and Function Signatures in λProlog

We distinguish λProlog types from ML types, which are values (or terms) in our λProlog implementation. Terms that represent ML types have the λProlog type `ml-type`. The clauses below define four λProlog constants: `int`, `bool`, `real`, and `string`. The first clause can be read as "the (λProlog) type of `int` is `ml-type`".

```
type int    ml-type.    % integer
type bool   ml-type.    % boolean
type real   ml-type.
type string ml-type.
```

In addition to these basic types, ML features several compound types. The first clause below declares the constant `list` to be a function that takes a term of type `ml-type` and produces a term of type `ml-type`. For example, `(list int)` forms a term representing the ML type for integer lists. The constant `fn` is a curried function and takes one argument at a time. Given two arguments it produces an `ml-type` term. The expression `(fn int int)` represents the ML type $int \rightarrow int$.

```
type list   ml-type -> ml-type.
type array  ml-type -> ml-type.
type fn     ml-type -> ml-type -> ml-type.    % function
type prod   ml-type -> ml-type -> ml-type.    % cartesian product
```

In order to build a signature library, we use the the constant `hasType` to form propositions that relate ML function identifiers with their ML type signatures. The predefined Lambda Prolog type `o` is the type of propositions.

```
type hasType ml-function -> ml-type -> o
```

Figure 1 gives λProlog clauses declaring the signatures of ML functions in our sample library. [1] For example, the ML plus function, denoted as `ml-plus`, has the ML type $int \times int \rightarrow int$. Since ML allows the use of type variables, we conveniently use λProlog variables to represent ML type variables. For example, the function `ml-hd` takes a list of elements of any type and returns a value of the element type. As opposed to the `ml-sort` function, the curried function `ml-gensort` takes a binary function on elements of any type and returns a function that takes a list of elements and returns a list of elements. Intuitively, `ml-gensort`'s functional argument is the comparision predicate to be used in sorting the elements in a list. Note that we can state type signatures for ML functions using `hasType` without explicitly declaring their Lambda Prolog types. Given their use in `hasType` rules, their types will be inferred to be `ml-function`.

```
hasType ml-not       (fn bool bool).
hasType ml-abs       (fn int int).                % Absolute value.
hasType ml-plus      (fn (prod int int) int).
hasType ml-lessthan  (fn (prod int int) bool).    % Less than.

hasType ml-hd        (fn (list A) A).
hasType ml-tl        (fn (list A) (list A)).
hasType ml-cons      (fn (prod A (list A)) (list A)).
hasType ml-null      (fn (list A) bool).
hasType ml-length    (fn (list A) int).
hasType ml-map       (fn (fn A B) (fn (list A) (list B))).
hasType ml-nth       (fn (prod (list A) int) A). % Returns nth element in list.
hasType ml-intsort   (fn (list int) (list int)). % Sorts lists of integers.
hasType ml-sort      (fn (list A) (list A)).     % Sorts lists of A.
hasType ml-gensort   (fn (fn (prod A A) bool) (fn (list A) (list A))
                                                 % Generic sorting function.
```

Figure 1: Signatures for a Library of ML Functions

[1] We prefix all ML function names with `ml-` to avoid conflict with predefined Lambda Prolog constants such as `not` and `div`.

3.2. Simple Signature Matching

Our queries ask "What functions have a signature that matches the pattern *S*?". Queries are solved by attempting to unify the signature *S* with the signature of each function *F* in the library. If unification with *F*'s signature is successful (a consistent binding of variables is found), then *F* satisfies the query.

To motivate our full signature matching predicate presented in the next section, we begin with some simple queries based on just our `hasType` predicate. We illustrate queries through scripts showing interaction with the λProlog system. The first line of each example begins with the λProlog prompt (`?-`) followed by a query. Results of the query follow on subsequent lines, where a single result is a binding of λProlog variables to values; multiple results to the same query are separated by a semicolon and the system replies `no` if there are no solutions.

The query below asks for all functions of type $bool \rightarrow bool$, of which there is only one, `ml-not`.

```
?- hasType F (fn bool bool).
 F == ml-not
```

The following query shows how variables can be instantiated in different ways to satisfy the query. In the first solution, `X` can be any `ml-type` term. In the last two solutions, specific values for `X` are given.

```
?- hasType F (fn (list X) X).
 F == ml-hd, X == X: ml-type ;
 F == ml-null, X == bool ;
 F == ml-length, X == int
```

The next example shows that we may miss a function that has an extra argument but otherwise matches the signature pattern. In this case the missing function is `ml-gensort`, which may be of interest when looking for sorting functions.

```
?- hasType F (fn (list int) (list int)).
 F == ml-tl ;
 F == ml-intsort ;
 F == ml-sort
```

The next two queries show that the order of arguments in the signature is significant, and that again relevant functions may be overlooked.

```
?- hasType F (fn (prod int (list X)) X).
 no
?- hasType F (fn (prod (list X) int) X).
 F == ml-nth, X == X: ml-type
```

3.3. Full Signature Matching

Recognizing that we may miss functions of interest through a naive use of the `hasType` predicate, we need a more general signature matching predicate that considers certain distinct signatures as "equivalent." Inspired by Rittri's [16] use of a congruence relation on signatures, which allow signatures with minor structural differences to match, we introduce the following functions that transform functions:

```
hasType ml-flip    (fn (fn (prod A B) C) (fn (prod B A) C)).
hasType ml-curry   (fn (fn (prod A B) C) (fn A (fn B C))).
hasType ml-uncurry (fn (fn A (fn B C)) (fn (prod A B) C)).
```

`Ml-flip` returns a function that reverses the order of the arguments of a function of two arguments. `Ml-curry` and `ml-uncurry` respectively curry and uncurry their functional arguments. Note that `ml-map` (Figure 1) is a transformer function as well.

We now define a signature matching predicate `satisfy-sig` that applies our transformation functions, thereby giving us more solutions. It explicitly uses the function `apply` to apply (transformation) functions to functions.

```
type satisfy-sig   ml-function -> ml-type -> int -> o.
type apply ml-function -> ml-function -> ml-function.

  satisfy-sig F T Depth :- hasType F T.                 % Base case.
  satisfy-sig (apply F G) T Depth :-                    % Recursive case.
      Depth > 0, NextDepth is (Depth - 1),
      satisfy-sig F (fn T1 T) NextDepth,
      satisfy-sig G T1 NextDepth.
```

The base case of `satisfy-sig` captures the notion of query satisfaction that we have been using so far. It states that a function `F` satisfies type signature pattern `T`, if `F` has a type signature that matches `T`. The recursive case makes use of our transformation functions by defining signature matching on applications of functions to functions. An application `(apply F G)` satisfies `T`, if `G` satisfies a signature `T1`, and `F` transforms functions with signature `T1` into functions with signature `T`. The third argument, `Depth`, ignored for the base case, is used in the recursive case to restrict the level of nested applications of `apply`.

Using `satisfy-sig` allows us to find the function we missed in an earlier query. Although the function `ml-nth` does not match the query, the function that results from swapping its arguments does match.

```
?- satisfy-sig Y (fn (prod int (list X)) X) 1.
 Y == apply ml-flip ml-nth, X == X: ml-type
```

The next query also shows that `satisfy-sig` can increase recall over using `hasType` alone.

```
?- satisfy-sig F (fn (list int) (list int)) 1.
 F == ml-tl ;
 F == ml-intsort ;
 F == ml-sort ;
 F == apply ml-gensort ml-lessthan ;
 F == apply ml-map ml-abs
```

But what about precision? Suppose in this last query we were looking for integer sorting functions. Then the fourth solution above is relevant, but the fifth is not. Function signatures do not carry enough information to distinguish these two results.

The clauses for `satisfy-sig` apply two functions that are inverses of one another (e.g., two flips, or curry and uncurry) and hence report trivially different solutions. In the example below, the Lambda Prolog system reports `yes` to indicate that it was interrupted by the user before it could report additional solutions to the query.

```
?- satisfy-sig F (fn (list int) (list int)) 3.
 F == ml-tl ;
 F == ml-intsort ;
 F == ml-sort ;
 F == apply ml-gensort ml-lessthan ;
 F == apply ml-gensort
        (apply ml-uncurry (apply ml-curry ml-lessthan)) ;
 F == apply ml-gensort (apply ml-flip ml-lessthan) ;
 F == apply ml-gensort
        (apply ml-flip (apply ml-flip ml-lessthan)) ;
 F == apply ml-map ml-abs ;
 F == apply (apply ml-curry (apply ml-uncurry ml-gensort))
        ml-lessthan ;
 F == apply (apply ml-curry (apply ml-uncurry ml-gensort))
        (apply ml-uncurry (apply ml-curry ml-lessthan)) ;
 F == apply (apply ml-curry (apply ml-uncurry ml-gensort))
        (apply ml-flip ml-lessthan) ;
 F == apply (apply ml-curry (apply ml-uncurry ml-gensort))
        (apply ml-flip (apply ml-flip ml-lessthan)) ;
 F == apply (apply ml-curry (apply ml-uncurry ml-map)) ml-abs ;
 yes
```

We can eliminate some of these extra solutions by modifying the `satisfy-sig` clauses to prohibit the application of inverses.

```
type invert ml-function -> ml-function -> o.
invert ml-flip ml-flip.
invert ml-curry ml-uncurry.
invert ml-uncurry ml-curry.
```

The revised definition of `satisfy-sig` is:

```
satisfy-sig F T Depth :- hasType F T.

satisfy-sig (apply F G) T D :-
    Depth > 0, NextDepth is (Depth - 1),
    satisfy-sig F (fn T1 T) NextDepth,
    satisfy-sig G T1 NextDepth,
    if (G = (apply H1 H2))
             (not (invert F H1))
             true.
```

where `if` is defined as:

```
type if  o -> o -> o -> o.
if P T E :- P, !, T.
if P T E :- E.
```

The only change to `satisfy-sig` is to add a test for whether the function we want to match, `G`, is the result of the application of two functions, a (transformation) function `H1` and `H2`; if it is then we match only if the transformation function `F` is not the inverse of `H1`. This additional check increases precision without decreasing recall. The last query will now result in fewer uninteresting solutions.

```
?- satisfy-sig F (fn (list int) (list int)) 3.
 F == ml-tl ;
 F == ml-intsort ;
 F == ml-sort ;
 F == apply ml-gensort ml-lessthan ;
 F == apply ml-gensort (apply ml-flip ml-lessthan) ;
 F == apply ml-map ml-abs
```

4. Matching Specifications

Signatures carry limited information for distinguishing functions in a library. For example, in a local ML library of 270 functions, over 50% have type $A \rightarrow B$ and 32% have type $A \times A \rightarrow B$, where A and B are ML base types. Of the latter, 22% have type either $int \times int \rightarrow int$ or $int \times int \rightarrow bool$. Using additional semantic information in queries could increase precision. In this section we extend the basic library search utility by introducing a mechanism to match specifications.

We write Larch-style specifications, each composed of two components: (1) An *interface component* describes individual program module behavior, e.g., the side effects of a Pascal procedure or exceptional termination of an ML function. It consists of a pre-condition and post-condition pair, each written as a first-order predicate. (2) A *shared component* defines the abstractions, e.g., properties of sets, lists, and partial orders, used in interface components. It consists of a set of algebraic equations that define relations among operators, and hence defines equality between terms that appear in an interface component.[2]

Below we present a small library of functions over collection-like objects defined in terms of lists. First, we give a shared component that contains clauses about list operators, and then, we give five interface components, each containing, in addition to a signature, a pre- and post-condition pair. These two subsections also show how we systematically transform the predicates and equations of Larch-style specifications into λProlog clauses. Finally, we give a set of clauses for solving queries that contain three parts: a signature, a pre-condition, and a post-condition.

4.1. Shared Component: A List Abstraction

Figure 2 gives a shared component specification for lists. The two constructors are `new` and `cons`.

```
type cons A -> (list A) -> (list A).
type new list A.
```

Using a set of λProlog propositions, we define each of the observers, `isempty`, `has`, `length`, and

[2]For this experiment, we ignore the inductive rules of inference that can appear in a Larch shared component.

`count` in terms of each of the constructors. The semantics of each boolean-valued observer can be given as a list of propositions that are true. For example, the first clause below for `isempty` states that "`isempty` on the `new` list is true."

```
type isempty list A -> o.
 isempty new.
 not (isempty (cons X Y)).
```

As in Prolog clauses, we read `:-` as reverse implication with the consequent on the left-hand side and the antecedent on the right-hand side. For example, in the second clause for `has`, an element, `E`, is in a non-empty list, `Y` if it was the last element, `X`, inserted or is already in the list `Y`.

```
type has list A -> A -> o.
 not (has new E).
 has (cons X Y) E :- if (E = X) true (has Y E).
```

The semantics of an observer that returns a non-boolean value is expressed as a proposition that relates arguments to results. The type of the Lambda Prolog constant `length` given below indicates the function it represents takes a list and produces an integer result. The first rule states that `length`, returns `0` given `new`. The second rule states that given `(cons X Y)`, the `length` function returns as the value of `L` the length of `Y` plus `1`.

```
type length list A -> int -> o.
 length new 0.
 length (cons X Y) L :- (length Y YLen), (L is (YLen + 1)).
```

In Figure 2, we define `count` similarly, where informally, the count function returns the number of occurrences a given element is in a given list.

4.2. Interface Component: Collection Functions

Figure 3 gives interface components for five functions over collection-like objects. For each function we provide its signature using a `hasType` clause, a `pre`-condition, and a `post`-condition. Given the following λProlog types,

```
type pre     ml-function -> (o -> o) -> o.
type post    ml-function -> (o -> o) -> o.
type with    A -> o -> o.
```

we see that each (pre- and post-) condition is a boolean-valued (λProlog) function. A pre-condition takes the same arguments as the (ML) function it describes. A post-condition takes those arguments plus one argument for each result. We use the constant `with` to express conditions in curried form. In λProlog λ`X.A` is written as `X \ A`. A `with` term contains a lambda expression that introduces a bound variable and has a predicate for a body. The bound variable represents one argument or one result of the (ML) function being defined. Since pre- and post-conditions are functions and we want to do reasoning with them, we need to use a higher-order logic in which functions are first-class objects. We will discuss the need for higher-order

```
type new list A.                        % The empty list.

type cons A -> (list A) -> (list A). % Add an element onto a list.

type isempty list A -> o.            % Is a list empty?
 isempty new.
 not (isempty (cons X Y)).

type has list A -> A -> o.           % Is an element in a list?
 not (has new E).
 has (cons X Y) E :- if (E = X) true (has Y E).

type length list A -> int -> o.      % What's the length of a list?
 length new 0.
 length (cons X Y) L :- (length Y YLen), (L is (YLen + 1)).

type count list A -> A -> int -> o.  % Occurrences of an element in a list.
 count new E 0.
 count (cons X Y) E C :- count Y E Part, if (E = X) (C is (Part + 1)) (C is Part).
```

Figure 2: Shared Component for List Abstraction

logic in detail in Section 5.1.

For example, `bagInit`'s signature indicates that it is a constant of ML type `bag`. Its post-condition states that the value of the resulting bag, when viewed as a list, is a list of length 0. (Note that the meaning of `length` is given by the shared component shown in Figure 2.)

```
hasType bagInit bag.
 pre bagInit (with B \true).
 post bagInit (with B \(length B 0)).
```

In the clauses for `bagAdd`, `B` denotes the value of the `bag` argument, `E` the integer argument, and `B2` the result. The pre-condition states that the bag argument must be represented by a list of length less than 100. The post-condition states that the result contains `E`.

```
hasType bagAdd (fn (prod bag int) bag).
 pre bagAdd
   (with B \(with E \((length B L), (less-than L 100)))).
 post bagAdd (with B \(with E \(with B2 \(has B2 E)))).
```

Looking at the clauses for the remaining functions, we note the following: (1) like `bagInit`, `containerInit` and `setInit` place no restrictions on its input argument (pre-condition is *true*), (2) the post-conditions for `bagInit`, `containerInit`, and `setInit` all differ, though intuitively they have the same meaning, and (3) unlike `bagAdd`, `setAdd`'s post-condition states that the result contains no duplicate elements.

```
hasType bagInit bag.                      % Initialize a bag.
  pre bagInit (with B \true).
  post bagInit (with B \(length B 0)).

hasType bagAdd (fn (prod bag int) bag).    % Insert an int in a bag.
  pre bagAdd (with B \(with E \((length B L), (less-than L 100)))).
  post bagAdd (with B \(with E \(with B2 \(has B2 E)))).

hasType containerInit container.          % Initialize a container.
  pre containerInit (with C \true).
  post containerInit (with C \((isempty C), (length C 0))).

hasType setInit set.                      % Initialize a set.
  pre setInit (with S \true).
  post setInit (with S \(isempty S)).

hasType setAdd (fn (prod set int) set).    % Insert an int in a set.
  pre setAdd (with S \(with E \true)).
  post setAdd (with S1 \(with E \(with S2 \((has S2 E),(count S2 E 1))))).
```

Figure 3: Interface Components for Five Functions

4.3. Full Specification Matching

A complete query now comprises three parts: a signature (and a depth), a pre-condition, and a post-condition. To satisfy a query these three parts of a function description must satisfy the three parts of a query.

```
type satisfies ml-function -> A -> o.
  satisfies F (query Sig Depth QPre QPost) :-
      satisfy-sig  F Sig Depth,
      satisfy-pre  F QPre,
      satisfy-post F QPost.
```

We defined satisfaction for signatures in Section 3. Here we define satisfaction for pre- and post-conditions. A pre-condition is satisfied if the pre-condition of the query implies the pre-condition of the library function. That is, the function's pre-condition can be weaker than the query's pre-condition, meaning that we can call the function in any context required by the query as well as other contexts. A post-condition is satisfied if the post-condition of the library function implies the post-condition of the query. That is, the function's post-condition can be stronger than the query's post-condition, meaning that the function may produce results for any context required by the query as well as other contexts. In λProlog, we capture these ideas as follows:

```
type satisfy-pre ml-function -> A -> o.
  satisfy-pre F QPre :- (pre F CPre), implies QPre CPre.

type satisfy-post ml-function -> A -> o.
  satisfy-post F QPost :- post F CPost, implies CPost QPost.
```

We define `implies` below. In λProlog, `pi` means "for all", and => means "implies".

```
type implies A -> B -> o.
  implies (with P) (with Q) :- !, (pi x \(implies (P x) (Q x))).
  implies P Q :- P => Q.
```

Recall that all conditions are represented by `with` terms, each of which takes a lambda expression argument. The first `implies` clause states that `implies` holds for `(with P)` and `(with Q)`, if it holds for all pairs of propositions resulting from identical substitution of type-correct values for the variables in lambda expressions `P` and `Q`. The second `implies` clause states that for simple propositions `implies` reduces to first-order implication. Our formulation of `implies` requires that for `implies` to hold between two propositions they must have the same number of `with` levels. This would be true of conditions for any functions with matching signatures.

Here now is an example of a full query and its solution. It asks for all functions `F` that take one argument of type T and another of type integer, and returns a result of type T. The query's pre-condition is just *true*[3] and its post-condition states that `F` should guarantee the result contains the integer argument. `SetAdd` is the only solution satisfying this query.

```
?- satisfies F (query (fn (prod T int) T) 1
      (with X \ (with Y \ true))
      (with X \ (with Y \ (with Z \ (has Z Y))))).
 F == setAdd, T == set
```

Suppose we were to satisfy the query's three parts separately:

```
?- satisfy-sig F (fn (prod T int) T) 1.
 F == ml-plus, T == int ;
 F == bagAdd, T == bag ;
 F == setAdd, T == set ;
 F == apply ml-flip ml-plus, T == int ;
 F == apply ml-flip ml-cons, T == list int

?- satisfy-pre F (with X \ (with Y \ true)).
 F == setAdd

?- satisfy-post F (with X \ (with Y \ (with Z \ (has Z Y)))).
 F == bagAdd ;
 F == setAdd
```

Note that the post-condition for `setAdd` is stronger than that of the query. In this particular example,

[3] Not all implementations of λProlog will allow `true` to appear as it does in our query.

changing the query's post-condition as follows has no effect on recall or precision.

```
?- satisfy-post F
     (with X \(with Y \(with Z \((has Z Y), (count Z Y 1))))).
 F == setAdd
?- satisfy-post F (with X \(with Y \(with Z \(count Z Y 1)))).
 F == setAdd
```

Looking back to the original query, we see that: (1) the signature query alone results in a set of functions many of which appear to be semantically unrelated, (2) unlike for the post-condition query alone, the function `bagAdd` is not a result of the original query because the query's pre-condition does not imply `bagAdd`'s pre-condition. In conclusion, additional semantic information increases precision dramatically.

5. Benefits and Limitations of λProlog

We are fortunate to have the power of higher-order unification provided by λProlog yet we do miss the power of (first-order) equational reasoning. We discuss each of these in turn.

5.1. Why Higher-order Logic?

To motivate our need for higher-order logic to do specification matching, let us consider expressing post-conditions within first-order logic. One way to do this is to represent a parameter in a post-condition by a λProlog variable. The post-conditions of the library functions would be expressed as:

```
post-1 bagInit (length B 0).
post-1 bagAdd (has B2 E).
post-1 containerInit ((isempty C), (length C 0)).
post-1 setInit (isempty S).
post-1 setAdd ((has S2 E), (count S2 E 1)).
```

We would define a `satisfy-post-1` predicate as:

```
satisfy-post-1 F Q :- post-1 F Y, Y => Q.
```

and pose the following query:

```
?- satisfy-post-1 G (has H J).
 G == bagInit
 H == cons X Y
 J == X ;
 yes
```

This query shows that λProlog will find substitutions for the variables `H` and `J` such that the query can be satisfied. Looking back on the rules for `has` we see that the term `(has (cons X Y) X)` reduces to `true`. Since `Y => true` for any `Y`, the query will match all functions in the library.

This formulation finds *particular* terms to substitute into query's post-condition variables. A correct formulation should allow the query to be satisfied only if it can be satisfied by *any* substitution in its post-condition. We can simulate modeling post-condition parameters not with Prolog variables, but with Prolog constants that have no clauses governing them. Consider each condition as a function with a list of parameters, where we name each parameter x*n*, with integer *n* representing its index in the parameter list. The post-conditions of the library functions would be expressed as:

```
post-2 bagInit (length x1 0).
post-2 bagAdd (has x3 x2).
post-2 containerInit ((isempty x1), (length x1 0)).
post-2 setInit (isempty x1).
post-2 setAdd ((has x3 x2), (count x3 x2 1)).
```

We would define `satisfy-post-2` as follows:

```
satisfy-post-2 F Q :- post-2 F Y, Y => Q.
```

Then for the following query,

```
?- satisfy-post-2 G (has x3 x2).
 G == bagAdd ;
 G == setAdd
```

we would get correct results. However, this second formulation has some limitations. The least important limitation is that the constants x*n* must be reserved and not used as function names. More significant limitations are on the reasoning that can be done with the post-condition functions.

The language we used to express the post-conditions is fairly simple and contains no binding constructs. If we introduce `let`, `for all`, or `there exists` constructs that create new variable bindings, we would have to introduce `implies` clauses to deal with post-condition functions that contained those constructs. Using first-order Prolog we would also have to express the variable substitution rules in order to handle these binding constructs properly. However, (higher-order) λProlog includes the substitution rules. Hence, we can express rules on any binding constructs without the need to give explicit clauses for substitution.

In summary, a higher-order Prolog can be simulated in first-order Prolog by encoding the variable substitution mechanism. By using higher-order Lambda Prolog, the rules are already correctly implemented. They are efficiently implemented since the substitution mechanism is integrated with unification.

Moreover, any additional reasoning about conditions we may want to do can be done directly in λProlog. An example of such reasoning would be flipping arguments in query pre- and post-conditions. Note that the order of parameters in query post-conditions is significant in the second first-order formulation (using constants for parameters) as well as in our higher-order formulation. Consider these queries:

```
?- satisfy-post-2 G (has x3 x1).
 no
?- satisfy-post G (with X1 \ (with X2 \ (with X3 \ (has X3 X1)))).
 no
```

In a manner similar to flipping the order of arguments when defining a more general signature matching, we can extend the higher-order formulation to flip the order of parameters in post-conditions.

```
satisfy-post (apply ml-flip F) QPost :-
    post F CPost, flip CPost CFlipped, implies CFlipped QPost.
```

where `flip` is defined as:

```
flip (with X \ (with Y \ ((P Y) X)))
     (with Y \ (with X \ ((P Y) X))).
```

Our query now has two solutions:

```
?- satisfy-post G (with X1 \ (with X2 \ (with X3 \ (has X3 X1)))).
 G == apply ml-flip bagAdd ;
 G == apply ml-flip setAdd
```

5.2. Equational Reasoning

We have seen that adding semantic information can dramatically increase precision. What effect does it have on recall? Consider the following post-condition queries.

```
?- satisfy-post F (with X \ (isempty X)).
 F == containerInit ;
 F == setInit
?- satisfy-post F (with X \ (length X 0)).
 F == bagInit ;
 F == containerInit
?- satisfy-post F (with X \ ((length X 0), (isempty X))).
 F == containerInit
```

Using equational reasoning we can deduce that `(length X 0) = (isempty X)`. However, λProlog does not use equational reasoning in solving queries. There may be library functions whose specifications are equivalent to that of the query, but expressed differently. Our search mechanism will miss those functions.

6. Related Work

Given a signature as a search key, doing an exact match will not always return all relevant components. To increase recall, we can relax the semantics of match to return all components whose signatures are in some sense either "equivalent" to or more "general" than the query's. Rittri [16] defines a congruence relation on types, whose formal justification is given in terms of Cartesian closed categories, that is used to match types "equivalent" to the key. Our inclusion of the transformation functions `flip`, `curry` and `uncurry` of Section 3 encode his equivalence relation. Runciman and Toyn [17] define a generality ordering on types, that with suitable restrictions can be turned into a partial order. Our use of an explicit `apply` in our definition of `satisfy-sig` gives us a similar effect of ordering types (and unifying over types rather than identifying them as in Rittri's case), e.g., that the type $A \rightarrow B$ is greater than the type B. To paraphrase Rittri [16], whereas his search method is based on the user's ignorance of argument order and currying, Runciman and Toyn's is based on the user's ignorance of extra arguments. Our solution combines both ideas into one framework.

To our knowledge, no previous work has been done on specification matching, where specifications capture formally the semantics of the objects they describe, e.g., in the form of pre- and post-condition predicates.

Tangentially related to our work on search is the reliance on dependency [3], hierarchy [15], and/or inheritance relations [6] among software components to *browse* through libraries. Specific examples include literate programming systems [8] where users attach informal documentation to code, and hypertext systems [2, 19] where users make explicit links between document parts. They focus on the relation between components rather than the components themselves, thereby facilitating general browsing, but not query-specific search.

7. Summary and Significance of Contributions, Future Work

The new idea we propose is to access software libraries using specifications as search keys. Specifically, we have shown that: (1) how to search based on signatures using first-order unification to do signature matching, and (2) how to search based on pre- and post-conditions using higher-order unification to do specification matching. One additional concrete contribution is our use of λProlog to specify ML library functions, following the Larch two-tiered specification style in particular.

Our work should be of interest to many communities involved in the traditional study of programming languages. First, for the logic programming community, we show a practical need for higher-order unification since it lets us do specification matching automatically. We show a practical use of λProlog as a specification and query language for our software library application and as an implementation language for our prototype search utility.

Second, for the formal specification and software engineering communities, we show a new use of specifications (as search keys), thus providing another incentive for specifying programs. Formal specifications have a long history of being useful in the software development process for program design and program verification. But what about software reuse? If software modules are truly to be reused either without change or for further tailoring, we cannot rely on identifying modules simply by name (and perhaps signature information) and then on "eyeballing" the code in the retrieved modules to see if any are relevant to our needs. We need to rely on module descriptions written in a language higher-level than the code itself. Hence, we see formal specifications as playing the key[4] role for software library search.

[4]Pun intended.

Third, programming language and database ideas are merging, as witnessed by programming language design influencing query language design (and vice versa), and more recently, the incorporation of persistence and atomic transactions in programming languages [9, 4, 1, 11]. We show a deeper connection between the two areas by identifying their different ideas of *satisfaction*: "a program *satisfies* a specification" and "a database object *satisfies* a query" are instances of the same general idea. Here, we use unification to do satisfaction checking, i.e., to do database retrieval.

Our experiment with λProlog shows the feasibility of our more general idea of specification matching; however, we do recognize that theoretical and practical challenges remain. For example, we used transformation functions like `flip` and an implicit ordering on functions to obtain higher recall. We could use different transformation functions and/or define different orderings on function signatures that would tradeoff precision and recall. Also, more generally, we could define orderings on specifications; viewing specifications as theories [20], for example, we could use theory inclusion as an ordering relation. We also observe that were we to have higher-order unification with equality then we would have a more expressive specification language. Work on combining first-order Horn clause logic with equality, e.g., as in Eqlog [5], is a step in that direction.

More practically, to realize the third stage of our pipelined query scenario, we encourage more work on improving the performance of current theorem provers. Finally, since more and bigger software libraries appear everyday [6, 14, 10], to make indexing and searching through them more effective, we encourage people to go through the trouble of attaching formal specifications to their software components.

8. Acknowledgments

This research was conducted in collaboration with members of the Venari and Ergo Projects at Carnegie-Mellon University. Thanks to Greg Morrisett, Linda Leibengood, and Scott Dietzen for commenting on an earlier draft of this paper. Greg Morrisett implemented the first version of signature matching in Turbo Prolog. Thanks also to Frank Pfenning for Lambda Prolog consulting.

References

[1] M. P. Atkinson, K. J. Chisolm, and W. P. Cockshott. PS-Algol: an Algol with a persistent heap. *SIGPLAN Notices*, 17(7):24–31, July 1982.

[2] E. J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 2(9):17–41, September 1987.

[3] F. DeRemer and H. H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Trans. on Soft. Eng.*, June 1976.

[4] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, pages 57–69, December 1988.

[5] J. A. Goguen and J. Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Functional and Logic Programming*, pages 179–210. Prentice-Hall, 1986.

[6] A. Goldberg and D. Robson. *Smalltalk 80: The Language and its Implementation*. Addison-Wesley, 1983.

[7] J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[8] D. E. Knuth. Literate programming. *Computer Journal*, 27(2):97–111, 1984.

[9] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Language and Systems*, 5(3):382–404, July 1983.

[10] Math. Mathematica. *Math*, January 1988.

[11] D. C. J. Matthews. Poly manual. *SIGPLAN Notices*, 20(9):52–76, September 1985.

[12] D. A. Miller and G. Nadathur. Higher-order logic programming. In *Third International Conference on Logic Programming*, London, July 1986.

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

[14] D. R. Musser and A. A. Stepanov. *The Ada Generic Library*. Springer-Verlag, 1989.

[15] D. L. Parnas. *On a 'Buzzword': Hierarchical Structure*, pages 336–339. North-Holland Publishing Company, 1974.

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

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

[18] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[19] J. B. Smith and S. F. Weiss. An overview of hypertext. *CACM*, 31(7), July 1988. See entire issue for related papers.

[20] W. M. Turski and T. S. E. Maibaum. *The Specification of Computer Programs*. Addison-Wesley, 1987.