

# Keyword Programming in Java

Greg Little · Robert C. Miller

Received: date / Accepted: date

**Abstract** Keyword programming is a novel technique for reducing the need to remember details of programming language syntax and APIs, by translating a small number of unordered keywords provided by the user into a valid expression. In a sense, the keywords act as a query that searches the space of expressions that are valid in the given context. Prior work has demonstrated the feasibility and merit of this approach in limited domains. This paper explores the potential for employing this technique in much larger domains, specifically general-purpose programming languages like Java. We present an algorithm for translating keywords into Java method call expressions. When tested on keywords extracted from existing method calls in Java code, the algorithm can accurately reconstruct over 90% of the original expressions. We tested the algorithm on keywords provided by users in a web-based study. The results suggest that users can obtain correct Java code using keyword queries as accurately as they can write the correct Java code themselves. We implemented the algorithm in an Eclipse plug-in as an extension to the autocomplete mechanism and deployed it in a preliminary field study of several users, with mixed results. One interesting result of this work is that most of the information in Java method call expressions lies in the keywords, and details of punctuation and even parameter ordering can often be inferred automatically.

**Keywords** Java · Autocomplete · Code Assistants

## 1 Introduction

Software development is rapidly changing and steadily increasing in complexity. Modern programmers must learn and remember the details of many programming languages and APIs in order to build and maintain today's systems. A simple web application

---

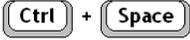
G. Little  
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139  
Tel.: +617-308-4673  
E-mail: glittle@gmail.com

R. Miller  
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139

```

public List<String> getLines(BufferedReader src) throws Exception {
    List<String> array = new ArrayList<String>();
    while (src.ready()) {
        add line
    }
    return array;
}

```



```

public List<String> getLines(BufferedReader src) throws Exception {
    List<String> array = new ArrayList<String>();
    while (src.ready()) {
        array.add(src.readLine());
    }
    return array;
}

```

**Fig. 1** In keyword programming, the user types some keywords, presses a completion command (such as Ctrl-Space in Eclipse), and the keywords are translated into a valid expression.

may require the use of half a dozen formal syntaxes – such as Java, Javascript, PHP, HTML, CSS, XML, SQL – in addition to many different APIs in each language. Learning, remembering, and using all these technologies correctly is becoming a significant burden.

Our goal is to develop techniques that reduce the burden of remembering the details of a particular language or API. The technique proposed in this paper, *keyword programming*, uses a few keywords provided by the user to search for expressions that are possible given the context of the code. The user interface takes the form of an advanced code completion interface in an IDE. For instance, Figure 1 shows a user entering **add line** in a Java file, which the system translates in-place to **lines.add(in.readLine())**. The generated expression contains the user’s keywords **add** and **line**, but also fills in many details, including the receiver objects **lines** and **in**, the full method name **readLine**, and the formal Java syntax for method invocation.

In this paper, we propose an algorithm for finding keyword query completions quickly. This work builds on keyword programming techniques in Chickenfoot [5] and Koala [4], but scales the algorithms to the larger domain of Java.

This work is similar to Prospector [6] and XSnippet [9], which suggest Java code given a return type and available types. However, our system uses keywords to guide the search, making it more expressive when type information alone is not enough to infer the desired code.

Our key contributions are:

- An algorithm for translating keyword queries into Java code efficiently.
- An evaluation of the algorithm on a corpus of open source programs, using artificial inputs generated by extracting keywords from existing method call expressions. The algorithm is able to reconstruct over 90% of the expressions correctly given only the keywords (in random order). This suggests that punctuation and ordering contribute relatively little information to most Java method calls, so an automatic technique like keyword programming can take care of these details instead.
- An evaluation of the algorithm on human-generated inputs. The algorithm translates the keyword queries with the same accuracy as users writing correct Java code without tool support (which is roughly 50% in our study).

- An Eclipse plug-in called Quack that provides a user interface for translating keyword queries in Java source files. This plug-in is available for people to download and try.
- A field test of Quack in which several developers incorporated the tool into their work flow for a week. The accuracy of the tool in the field study was similar to the earlier human-generated input study (roughly 50%), and the study also yielded some important lessons about its user interface.

In the next section, we present a model, which is followed by a problem statement. We then present the algorithm, and two evaluations of the algorithm itself. This is followed by a presentation of an Eclipse plug-in that provides an interface to the algorithm. We also discuss a field test of the plug-in. Then we discuss related work, future work, and conclusions.

## 2 Model

We want to model the following scenario: a user is at some location in their source code, and they have entered a keyword query. The keywords are intended to produce a valid expression in a programming language, using APIs and program components that are accessible at that point in the source code. In order to find the expression, we need to model the context of the location in the source code. In the case of Java, we need to model the available methods, fields and local variables, and how they fit together using Java’s type system. The resulting model defines the search space.

Although this paper focuses on Java, our model is more generic, and could be applied to many languages. We define the model  $M$  as the triple  $(T, L, F)$ , where  $T$  is a set of types,  $L$  is a set of labels used for matching the keywords, and  $F$  is a set of functions.

### 2.1 Type Set: $T$

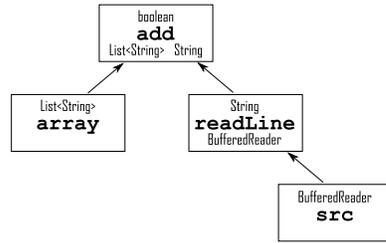
Each type is represented by a unique name. For Java, we get this from the fully qualified name for the type. Examples include `int` and `java.lang.Object`.

We also define  $sub(t)$  to be the set of both direct and indirect subtypes of  $t$ . This set includes  $t$  itself, and any type that is assignment-compatible with  $t$ . We also include a universal supertype  $\top$ , such that  $sub(\top) = T$ . This is used when we want to place no restriction on the resulting type of an expression.

### 2.2 Label Set: $L$

Each label is a sequence of alphanumeric keywords. We use labels to represent method names, so that we can match them against the keywords in a query.

To get the keywords from an identifier, we break up the identifier at punctuation and capitalization boundaries. For instance, the method name `currentTimeMillis` is represented with the label **(current, time, millis)**. Note that capitalization is ignored when labels are matched against the user’s keywords.



**Fig. 2** This is a function tree representing the Java expression `array.add(src.readLine())` from Figure 1. Each node is associated with a function. The words at the top of each node represent the return type, and the words at the bottom represent the parameter types. Note that `array` and `src` are modeled as functions, even though they are local variables in the source code.

### 2.3 Function Set: $F$

Functions are used to model each component in an expression that we want to match against the user’s keyword query. In Java, these include methods, fields, and local variables.

We define a function as a tuple in  $T \times L \times T \times \dots \times T$ . The first  $T$  is the return type, followed by the label, and all the parameter types. As an example, the Java function: `String toString(int i, int radix)` is modeled as `(java.lang.String, (toString), int, int)`.

For convenience, we also define  $ret(f)$ ,  $label(f)$  and  $params(f)$  to be the return type, label, and parameter types, respectively, of a function  $f$ .

### 2.4 Function Tree

The purpose of defining types, labels and functions is to model expressions that can be generated by a keyword query. We model expressions as a function tree. Each node in the tree is associated with a function from  $F$ , and obeys certain type constraints.

As an example, Figure 2 shows a function tree modeling the expression `array.add(-src.readLine())` from Figure 1. Note that `array` and `src` are modeled as functions, even though they are local variables in the source code. This is discussed more in the Java Mapping section below.

In order to talk about function trees more formally, we define each node as a tuple consisting of an element from  $F$  followed by some number of child nodes. For a node  $n$ , we define  $func(n)$  to be the function, and  $children(n)$  to be the list of child nodes. We require that the number of children in a node be equal to the number of parameter types of the function, i.e.,  $|children(n)| = |params(func(n))|$ . This constraint requires that the `add` node in Figure 2 have two children, because `add` has two parameters. We also require that the return types from the children fit into the parameters, i.e.,  $\forall_i ret(func(children(n)_i)) \in sub(params(func(n))_i)$ . This constraint requires that the `readLine` function in Figure 2 return the type `String` (or a subtype of `String`), because the second parameter of `add` is `String`.

Note that in the end, the system renders the function tree as a syntactically-correct and type-correct expression in the underlying language (in this case Java). The function tree in Figure 2 is rendered as `array.add(src.readLine())`.

---

## 2.5 Java Mapping

We now provide the particulars for mapping various Java elements to  $T$ ,  $L$  and  $F$ . Most of these mappings are natural and straightforward, and could be adapted to other languages.

### 2.5.1 Classes

A class or interface  $c$  is modeled as a type in  $T$ , using its fully qualified name (e.g. `java.lang.String`). Any class that is assignment compatible with  $c$  is added to  $sub(c)$ , including any classes that extend or implement  $c$ .

The model includes all classes that are directly referenced in the current source file, plus classes that can be obtained from those classes by method calls or field references. Since the function trees generated by our algorithm are bounded by a maximum depth, the model does not include classes that can only be obtained by a method call sequence longer than that maximum.

### 2.5.2 Primitive Types

Because of automatic boxing and unboxing in Java 1.5, we model primitive types like `int`, and `char` as being the same as their object equivalents `java.lang.Integer` and `java.lang.Character`.

### 2.5.3 Instance Methods

Instance methods are modeled as functions that take their receiver object as the first parameter. For instance, the method: `public Object get(int index)` of `Vector` is modeled as: `(java.lang.Object, (get), java.util.Vector, int)`.

### 2.5.4 Fields

Fields become functions that return the type of the field, and take their object as a parameter. For instance, the field `public int x` of `java.awt.Point` is modeled as: `(int, (x), java.awt.Point)`.

### 2.5.5 Local Variables

Local variables are simply functions that return the type of the variable and take no parameters, e.g., the local variable `int i` inside a `for`-loop is modeled as `(int, (i))`.

### 2.5.6 Constructors

Constructors are modeled as functions that return the type of object they construct. We use the keyword `new` and the name of the class as the function label, e.g., the constructor for `java.util.Vector` that takes a primitive `int` as a parameter is represented by: `(java.util.Vector, (new, vector), int)`.

### 2.5.7 Members of Enclosing Class

Member methods and fields of the class containing the keyword query are associated with an additional function, to support the Java syntax of accessing these members with an assumed `this` token. The new function doesn't require the object as the first parameter. For instance, if we are writing code inside `java.awt.Point`, we would create a function for the field `x` like this: `(int, (x))`. Note that we can model the keyword `this` with the additional function `(java.awt.Point, (this))`.

### 2.5.8 Statics

In Java, a static method or field is normally accessed using its class name, but can also be called on a receiver object. To support both forms, we use two functions. For instance `static double sin(double a)` in `java.lang.Math` is modeled with both: `(double, (sin), java.lang.Math, double)`, and `(double, (math, sin), double)`.

Note that in the second case, `math` is included in the function label. This is done since experienced Java programmers are used to including the type name when calling static methods.

### 2.5.9 Generics

The model represents generic types explicitly, i.e., we create a new type in  $T$  for each instantiation of a generic class or method. For instance, if the current source file contains a reference to both `Vector<String>` and `Vector<Integer>`, then we include both of these types in  $T$ . We also include all the methods for `Vector<String>` separately from the methods for `Vector<Integer>`. For example, the model includes both of the following the `get` methods: `(String, (get), Vector<String>, int)`, and `(Integer, (get), Vector<Integer>, int)`.

The motivation behind this approach is to keep the model simple and programming-language-agnostic. In practice, it does not explode the type system too much, since relatively few different instantiations are visible at a time.

### 2.5.10 Other Mappings

We have experimented with additional mappings of Java syntax into our model. In the Eclipse plug-in described in section 8, we include numeric and string literals, as well as array indexing. In section 11.1, we consider other extensions, such as variable assignment and control flow syntax. Implementing and evaluating these extensions is future work.

## 3 Problem

Now that we have a model of the domain, we can articulate the problem that our algorithm must solve.

The input to the algorithm consists of a model  $M$  and a keyword query. We also supply a desired return type, which we make as specific as possible given the source code around the keyword query. If any type is possible, we supply  $\top$  as the desired return type.

The output is a valid function tree, or possibly more than one. The root of the tree must be assignment-compatible with the desired return type, and the tree should be a good match for the keywords according to some metric.

Choosing a good similarity metric between a function tree and a keyword query is the real challenge. We need a metric that matches human intuition, as well as a metric that is easy to evaluate algorithmically.

Our metric is based on the simple idea that each input keyword is worth 1 point, and a function tree earns that point if it “explains” the keyword by matching it with a keyword in the label of one of the functions in the tree. This scoring metric is described in more detail in the next section.

## 4 Algorithm

The algorithm uses a score metric to measure the similarity of a function tree to the keyword query. Function trees with a higher score are considered more similar.

The score for a function tree is dominated by the number of keywords that match the keyword query, but we have experimented with other heuristics. Given a function tree where  $T_{funcs}$  is the list of functions in the tree, we calculate the score as follows:

- +1.0 for each keyword  $k$  in the query where there exists an  $f \in T_{funcs}$  such that  $k \in label(f)$ . This gives 1 point for each keyword that the tree explains.
- -0.05 for each  $f \in T_{funcs}$ . This favors function trees with fewer functions.
- For each  $f \in T_{funcs}$ , consider each  $w \in label(f)$ , and subtract 0.01 if  $w$  is not in the query. This favors less verbose function trees.
- +0.001 for each  $f \in T_{funcs}$  where  $f$  is a local variable, or  $f$  is a member variable or member method of the enclosing class. This favors functions and variables which are close to the user’s context.

We would like the property that the score metric for a function tree is simply the sum of some value associated with each function in the tree. This gives us a clean way to add additional heuristics. To achieve this, we associate a vector with each function, and invent a special way of adding these vectors together so that their sum has the desired property. We call these *explanation vectors*, and they are explained next.

### 4.1 Explanation Vector

If we have  $n$  keywords  $k_1, k_2, \dots, k_n$  in the input query, then an explanation vector has  $n + 1$  elements  $e_0, e_1, e_2, \dots, e_n$ . Each element  $e_i$  represents how well we have explained the keyword  $k_i$  on a scale of 0 to 1; except  $e_0$ , which represents explanatory power not associated with any particular keyword. When we add two explanation vectors together, we cap the resulting elements associated with keywords ( $e_1, e_2, \dots, e_n$ ) at 1, since the most we can explain a particular keyword is 1. Note that we do not cap the resulting element  $e_0$ . Explanation vectors are compared by summing each vector’s elements to produce a scalar score, and then comparing those scores.

We calculate an explanation vector  $expl(f)$  for each function  $f \in F$ . In the common case, we set  $e_i$  to 1 if  $label(f)$  contains  $k_i$ . For instance, if the input is:

**is queue empty**

and the function  $f$  is `(boolean, (is, empty), List)`, then  $expl(f)$  would be:

`(e0, 1is, 0queue, 1empty)`

Now consider the input:

**node parent remove node**

where `node` is a local variable modeled with the function `(TreeNode, (node))`. Since `node` appears twice in the input, we distribute our explanation of the word `node` between each occurrence:

`(e0, 0.5node, 0parent, 0remove, 0.5node)`

In general, we set  $e_i = \max(\frac{x}{y}, 1)$ , where  $x$  is the number of times  $k_i$  appears in  $label(f)$ , and  $y$  is the number of times  $k_i$  appears in the input.

To calculate  $e_0$ , we start by setting it to  $-0.05$ . This accounts for the notion that each function call in a function tree adds  $-0.05$  to the total score. Next, we subtract  $0.01$  from  $e_0$  for each word in  $label(f)$  that does not appear in the input query. Finally, we add  $0.001$  if this function represents a local variable, member variable or member method.

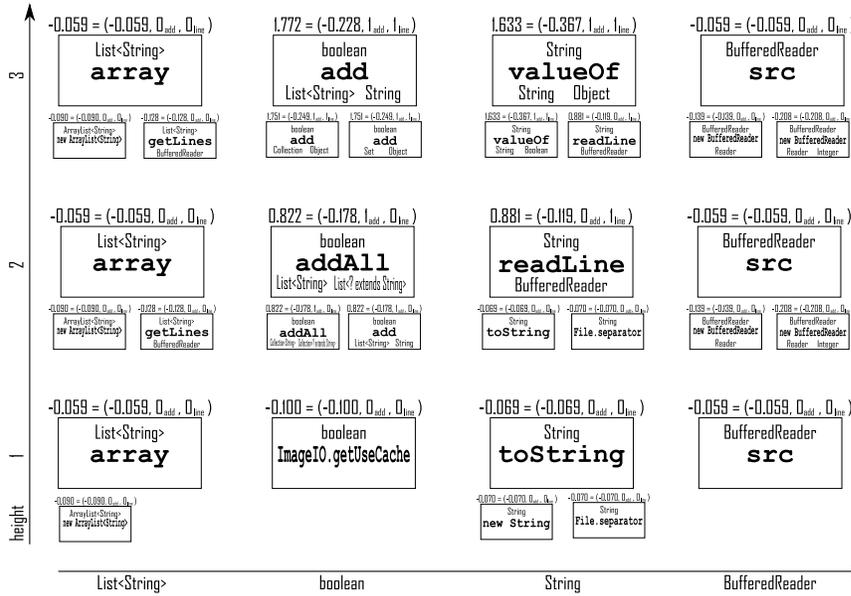
## 4.2 Dynamic Program

Our algorithm uses a dynamic program, which consists of a table of the following form: each column represents a return type (for every  $t \in T$ ), and each row represents a maximum tree height (ranging from 1 to  $h$ ). Each cell contains at most  $r$  pairs. Each pair contains a cumulative explanation vector, and a root function. This explanation vector encodes the explanatory power of the best known function tree rooted at the given function, and with tree height bounded by the maximum height for the given row. We identify the pairs in the cell with return type  $t$  and maximum tree height  $i$  as  $bestRoots(t, i)$ . Figure 3 shows a sample run of the dynamic program on the input **add line** in the context of Figure 1. In the figure,  $bestRoots(BufferedReader, 1)$  contains the pair  $((-0.059, 0_{add}, 0_{line}), (BufferedReader, (src)))$ .

Note that this dynamic program is parameterized by a couple of factors: the number of rows  $h$ , and the number of functions in each cell  $r$ . In our experiments, we set  $h = 3$  and  $r = 3$ . These values are reflected in Figure 3: there are three rows of cells, and each cell contains at most three function boxes. A height of three seemed large enough to build common Java expressions. We chose three as a starting point for experimentation with  $r$ . We did increase  $r$  to five in the special case of the column for `java.lang.Object`, since many functions return subtypes of this type. Ultimately we need to experiment more with both of these factors, but thus far, they do not appear to be the bottleneck of the algorithm.

The dynamic program fills out the table row by row, starting with height 1 and going up to  $h$ . In each row, we consider each type, and for each type, we consider each function which returns the type or a subtype. When we consider a function, we want to find the best explanation vector of a function tree with this function at the root. This is done by summing the explanation vector for the function itself with explanation vectors for each parameter, drawn from lower rows in the table. Pseudocode for this process is shown in Figure 4.

To illustrate some of this computation, consider the `readLine` function box at height 2 in Figure 3. The explanation vector associated with this function is  $(-0.119, 0_{add},$



**Fig. 3** The dynamic programming table for a run of the algorithm on the input **add line** in the context of Figure 1. Each row represents a height, and each column represents a return type. Each cell contains up to three boxes. The box with the highest score in each cell is enlarged. Each box represents a function with the name in the center, the return type above the name, and the parameter types listed below the name. The return type of each function must be a subtype of the column type that the box is in. The numbers in parenthesis above each function represent the cumulative explanation vector of the best known function tree rooted at that function. To see that the explanation vectors are cumulative, notice that the explanation vector for the **add** function in the top row has a  $1_{add}$  and a  $1_{line}$ , even though the **add** function itself does not contain both keywords **add** and **line**. The  $1_{line}$  is coming from the explanation vector for **readLine**, which is the best option for satisfying the **String** parameter at height 2.

$1_{line}$ ). To compute this, we start with the explanation vector for the function **readLine**, which is  $(-0.06, 0_{add}, 1_{line})$ . The  $-0.06$  comes from  $-0.05$  as a baseline, and  $-0.01$  for the keyword **read** which does not appear in the input query **add line**. Now since **readLine** requires a parameter of type **BufferedReader**, the algorithm looks in the **BufferedReader** column at height 1 and finds **src** with explanation vector  $(-0.059, 0_{add}, 0_{line})$ . We add these explanation vectors together to get  $(-0.119, 0_{add}, 1_{line})$ , which is what we see in the figure.

### 4.3 Extraction

After we have run the dynamic program, we need to extract the highest-scoring function tree. To do this, we use a greedy recursive algorithm (see Figure 5) which takes the following parameters: a desired return type  $t$ , a maximum height  $h$ , and an explanation vector  $e$  (representing what we have explained so far). The function returns a new function tree, and an explanation vector.

```

procedure DYNAMICPROGRAM()
  /* this function modifies bestRoots */
  for each  $1 \leq i \leq h$ 
    for each  $t \in T$ 
      do {
         $bestRoots(t, i) \leftarrow \emptyset$ 
        for each  $f \in F$  where  $ret(f) \in sub(t)$ 
          do {
             $e \leftarrow GETBESTEXPLFORFUNC(f, i - 1)$ 
            if  $e > -\infty$ 
              then {  $bestRoots(t, i) \leftarrow bestRoots(t, i) \cup (e, f)$ 
                /* keep the best  $r$  roots */
                 $bestRoots(t, i) \leftarrow GETBESTN(bestRoots(t, i), r)$ 
              }
          }
      }

procedure GETBESTEXPLFORFUNC( $f, h_{max}$ )
   $e_{cumulative} \leftarrow expl(f)$ 
  for each  $p \in params(f)$ 
     $e_{best} \leftarrow (-\infty, 0, 0, 0, \dots)$ 
    for each  $1 \leq i \leq h_{max}$ 
      do {
        for each  $(e', f') \in bestRoots(p, i)$ 
          do {
            if  $e_{cumulative} + e' > e_{best}$ 
              then {  $e_{best} \leftarrow e_{cumulative} + e'$ 
            }
          }
         $e_{cumulative} \leftarrow e_{best}$ 
      }
  return ( $e_{cumulative}$ )

```

**Fig. 4** Pseudocode to fill out the dynamic programming table.

Note that we sort the parameters such that the most specific types appear first ( $t_1$  is more specific than  $t_2$  if  $|sub(t_1)| < |sub(t_2)|$ ). We do this because more specific types are likely to have fewer options, so the greedy algorithm is better off making a choice for them first. This idea is best explained with an example: let's say our query is **find roots**, and we are considering a function with parameter types  $p_1$  and  $p_2$ . Let's also say that  $p_2$  is a very specific type, and the only explanation vector associated with it explains **roots**. However,  $p_1$  is not very specific, and it has two options: one option explains **find**, and the other option explains **roots**. If we use our greedy algorithm, and we consider these parameters in order, then we might choose the option for  $p_1$  that explains **roots**, since both options increase our score by an equal amount. If instead we consider  $p_2$  first, then we would choose the option that explains **roots** for  $p_2$ , and the algorithm would know to choose the option for  $p_1$  that explains **find**, since **roots** is already explained.

#### 4.4 Running Time

Assume the user enters  $n$  keywords; in a preprocessing step, we spend  $O(|F|n)$  time calculating the explanation vector for each function against the keywords. We can approximate the running time of the dynamic program in Figure 4 by taking the product of all the for-loops, including the for-loops in *GetBestExplForFunc*. If we assume that every function takes  $p$  parameters, then we get  $O(h|T||F|p^hr)$ .

---

```

procedure EXTRACTTREE( $t, h, e$ )
 $e_{best} \leftarrow 0$ 
 $n_{best} \leftarrow null$ 
for each  $1 \leq i \leq h$ 
  for each  $(e_{ignore}, f) \in bestRoots(t, i)$ 
    /* create tuple for function tree node */
     $n \leftarrow (f)$ 
     $e_n \leftarrow e + expl(f)$ 
    /* put most specific types first */
     $P \leftarrow SORT(params(f))$ 
    for each  $p \in P$ 
      do
         $n_p, e_p \leftarrow EXTRACTTREE(p, i - 1, e_n)$ 
        /* add  $n_p$  as a child of  $n$  */
         $n \leftarrow append(n, n_p)$ 
         $e_n \leftarrow e_n + e_p$ 
        if  $e_n > e_{best}$ 
          then
             $e_{best} \leftarrow e_n$ 
             $n_{best} \leftarrow n$ 
return  $(n_{best}, e_{best})$ 

```

**Fig. 5** Pseudocode to extract a function tree.

We can do a little better by noting that if we cache the return values of *GetBestExplForFunc*, then we need to call this function at most  $|F|$  times for each row. We can make all of these calls for a given row, before iterating over the types, in  $O(|F|p hr)$  time. This gives us  $O(h(|F|p hr + |T||F|))$  total time.

Next, we observe that the loop over functions for each type only includes functions which return a subtype of the given type. For a type like `java.lang.Object`, this is likely to be all  $|F|$  functions, but on average this number is likely to be much less. If we assume this number averages to  $f$ , then we get  $O(h(|F|p hr + |T|f))$ .

Extracting the best function tree requires an additional  $O((hrp)^h)$  time, based on the recursive function in 5, which has a recursive depth of at most  $h$ .

In practice, the algorithm is able to generate height-3 function trees in well under a second with thousands of functions in  $F$ , hundreds of types in  $T$ , and a dozen keywords. More detailed information is provided in the evaluation that follows.

## 5 Evaluations

We conducted two evaluations of the algorithm. The first evaluation used artificially generated keyword queries from open source Java projects. This evaluation gives a feel for the accuracy of the algorithm, assuming the user provides only keywords that are actually present in the desired expression. It also provides a sense for the speed of the algorithm given models generated from contexts within real Java projects.

The second evaluation looks at the accuracy of the algorithm on human generated inputs; these inputs were solicited from a web survey, where users were asked to enter pseudocode or keywords to suggest a missing Java expression.

**Table 1** Project Statistics

Project	Class Files	LOC	Test Sites
Azureus	2,277	339,628	82,006
Buddi	128	27,503	7,807
CAROL	138	18,343	2,478
Dnsjava	123	17,485	2,900
Jakarta CC	41	10,082	1,806
jEdit	435	124,667	25,875
jMemorize	95	1,4771	2,604
Jmol	281	88,098	44,478
JRuby	427	72,030	19,198
Radeox	179	10,076	1,304
RSSOwl	201	71,097	23,685
Sphinx	268	67,338	13,217
TV-Browser	760	119,518	29,255
Zimbra	1,373	256,472	76,954

## 6 Artificial Corpus Study

We created a corpus of artificial keyword queries by finding expressions in open source Java projects, and obfuscating them (removing punctuation and rearranging keywords). We then passed these keywords to the algorithm, and recorded whether it reconstructed the original expression. This section describes the results.

### 6.1 Projects

We selected 14 projects from popular open source web sites, including sourceforge.net, codehaus.org, and objectweb.org. Projects were selected based on popularity, and our ability to compile them using Eclipse. Our projects include: *Azureus*, an implementation of the BitTorrent protocol; *Buddi*, a program to manage personal finances and budgets; *CAROL*, a library for abstracting away different RMI (Remote Method Invocation) implementations; *Dnsjava*, a Java implementation of the DNS protocol; *Jakarta Commons Codec*, an implementation of common encoders and decoders; *jEdit*, a configurable text editor for programmers; *jMemorize*, a tool involving simulated flashcards to help memorize facts; *Jmol*, a tool for viewing chemical structures in 3D; *JRuby*, an implementation of the Ruby programming language in Java; *Radeox*, an API for rendering wiki markup; *RSSOwl*, a newsreader supporting RSS; *Sphinx*, a speech recognition system; *TV-Browser*, an extensible TV-guide program; and *Zimbra*, a set of tools involving instant messaging.

Table 1 shows how many class files and non-blank lines of code each project contains. The table also reports the number of possible test sites, which are defined in the next section.

### 6.2 Tests

Each test is conducted on a method call, variable reference or constructor call. We only consider expressions of height 3 or less, and we make sure that they involve only the Java constructs supported by our model. For example, these include local variables and static fields, but do not include literals or casts. We also exclude expressions inside of

```

public IRubyObject callMethod(RubyModule context, String name, IRubyObject[] args,
    CallType callType) {
    :
    :
    if (method.isUndefined() ||
    :
    :
        IRubyObject[] newArgs = new IRubyObject[args.length + 1];
        System.arraycopy(args, 0, newArgs, 1, args.length);
        newArgs[0] = RubySymbol.newSymbol(getRuntime(), name);

        return callMethod("method_missing", newArgs);
    }
    :
    :
}

```

Fig. 6 Example Test Site

inner classes since it simplifies our automated testing framework. Finally, we discard test sites with only one keyword as trivial.

Figure 6 shows a test site in the JRuby project. This example has height 2, because the call to `getRuntime()` is nested within the call to `newSymbol()`. Note that we count nested expressions as valid test sites as well, e.g., `getRuntime()` in this example would be counted as an additional test site.

To perform each test, we obfuscate the expression by removing punctuation, splitting camel-case identifiers, and rearranging keywords. We then treat this obfuscated code as a keyword query, which we pass to the algorithm, along with a model of the context for the expression. If we can algorithmically infer the return type of the expression based solely on context, then we give the algorithm this information as well.

For example, the method call highlighted in Figure 6 is obfuscated to the following keyword query: **name runtime get symbol symbol ruby new**

The testing framework observes the location of this command in an assignment statement to `newArgs[0]`. From this, it detects the required return type:

```
org.jruby.runtime.builtin.IRubyObject
```

The framework then passes the keyword query and this return type to the algorithm. In this example, the algorithm returns the Java code:

```
RubySymbol.newSymbol(getRuntime(), name)
```

We compare this string with the original source code (ignoring whitespace), and since it matches exactly, we record the test as a success.

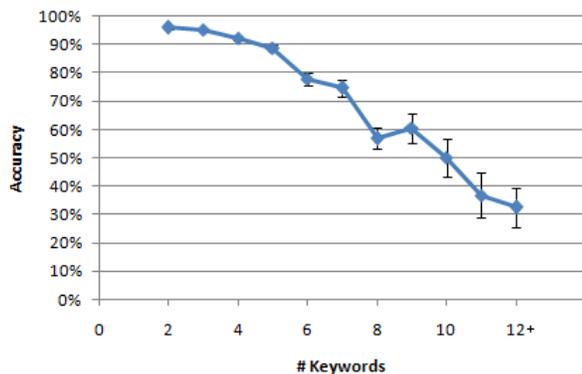
We also recorded other information about each test, including the length of the keyword query, the size of the model that was searched ( $|T|$  and  $|F|$ ), and the total time spent searching. Measured time did not include the time taken to construct the model, since in practice the model could be constructed once and reused for many queries. The test framework was implemented as a plug-in for Eclipse 3.2 with Java 1.6, and ran on an AMD Athlon X2 (Dual Core) 4200+ with 1.5GB RAM. The algorithm implementation was single threaded.

### 6.3 Results

The results presented here were obtained by randomly sampling 500 test sites from each project (except Zimbra, which is really composed of 3 projects, and we sampled

**Table 2** Number of test sites by keyword query length.

# Keywords	Samples
2	3330
3	1997
4	1045
5	634
6	397
7	206
8	167
9	86
10	54
11	38
$\geq 12$	46

**Fig. 7** Accuracy of algorithm, measured by the percentage of keyword queries that correctly reproduced the original expression, as a function of keyword query length. Error bars show standard error.

500 from each of them). This gives us 8000 test sites. For each test site, we ran the procedure described above.

Table 2 shows how many samples we have for different keyword query lengths. Because we do not have many samples for large lengths, we group all the samples of length 12 or more when we plot graphs against keyword length.

Figure 7 shows the accuracy of the algorithm given a number of keywords. The overall accuracy is 91.2%, though this average is heavily weighted to inputs with fewer keywords, based on the sample sizes shown in Table 2.

Figure 8 shows how long the algorithm spent processing inputs of various lengths. The average running time is under 500 milliseconds even for large inputs.

Another factor contributing to running time is the size of  $T$  and  $F$  in the model. Table 9 shows the average size of  $T$  and  $F$  for each project. The average size of  $F$  tends to be much larger than  $T$ . Figure 10 shows running time as a function of the size of  $F$ . We see that the algorithm takes a little over 1 second when  $F$  contains 14,000 functions. We believe that the size of  $F$  may explain the dip seen at the end of Figure 8: the average size of  $F$  in the model for inputs of 11 keywords was 4255, whereas the average size of  $F$  for inputs of 12 or more keywords was 3778.

We ran another experiment on the same corpus to measure the performance of the algorithm when given *fewer* keywords than were found in the actual expression,

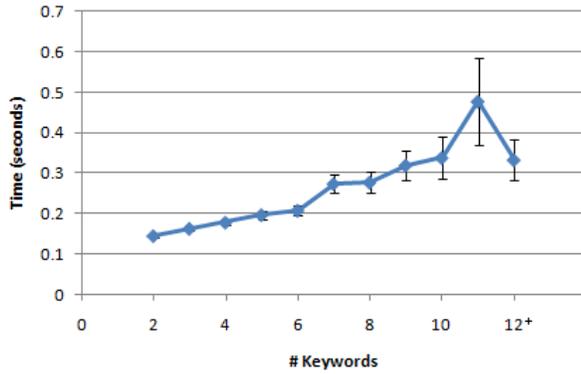


Fig. 8 Running time as a function of keyword query length. Error bars show standard error.

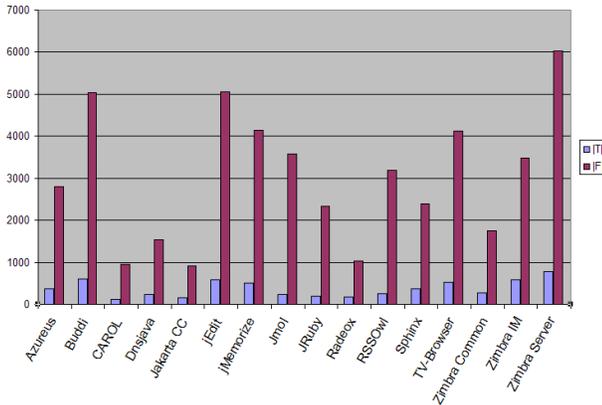


Fig. 9 Average size of  $T$  and  $F$  for different projects.

forcing it to infer method calls or variable references without any keyword hint. This experiment considered only test sites that were nested expressions (i.e. the resulting function tree had at least two nodes), so that when only one keyword was provided, the algorithm would have to infer at least one function to construct the tree.

Again, we randomly sampled 500 test sites from each project. At each test site, we first ran the algorithm with the empty string as input, testing what the algorithm would produce given only the desired return type. Next, we chose the most unique keyword in the expression (according to the frequency counts in  $L$ ), and ran the algorithm on this. We kept adding the next most unique keyword from the expression to the input, until all keywords had been added. The left side of Figure 11 shows the number of keywords we provided as input. The table shows the accuracy for different expression lengths (measured in keywords).

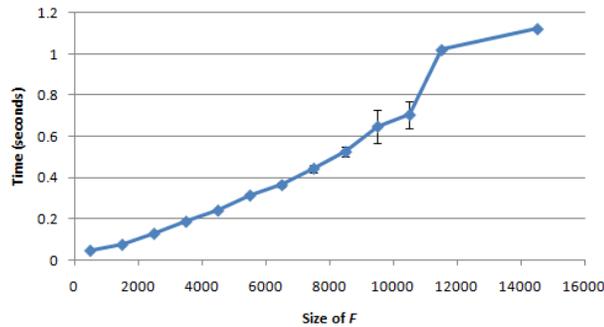


Fig. 10 Time given size of  $F$ . Error bars show standard error.

	Keywords in expression						
	2	3	4	5	6	7	8
0	0.042	0.032	0.028	0.041	0.009	0.012	0.007
1	0.348	0.271	0.239	0.169	0.108	0.073	0.031
2	0.964	0.77	0.562	0.405	0.284	0.26	0.139
3		0.946	0.796	0.647	0.487	0.421	0.322
4			0.895	0.777	0.609	0.467	0.36
5				0.834	0.704	0.593	0.435
6					0.75	0.638	0.6
7						0.683	0.596
8							0.623

Fig. 11 Accuracy of inference (1 is 100%)

#### 6.4 Discussion

The goal of the artificial corpus study was to establish feasibility: whether keyword programming could be done at all in Java, or if the search space was simply too big. The results suggest that the problem is tractable, and that a simple algorithm can achieve acceptable speed and modest accuracy.

The speed is reasonable for an Eclipse autocomplete-style plug-in; many queries are resolved in less than 500 milliseconds, provided that the model for the scope of the query is not too big (under 8000 functions). Note that we didn't include the time it takes to build the model in these measurements, since the model can be constructed in the background, before the user submits a query. Constructing the model from scratch can take a second or more, but a clever implementation can do better by updating the model as the user writes new code.

The accuracy on artificial inputs is encouraging enough to try the algorithm on user generated queries, which we do in the next section.

### 7 Web User Study

The purpose of this study was to test the robustness of the algorithm on human generated inputs. Inputs were collected using a web based survey targeted at experienced Java programmers.

---

### 7.0.1 Participants

Subjects were solicited from a public mailing list at a college campus, as well as a mailing list directed at the computer science department of the same college. Participants were told that they would be entered into a drawing for \$25, and one participant was awarded \$25.

Sixty-nine people participated in the study, but users who didn't answer all the questions, or provided obviously garbage answers such as "dghdfghdf...", were removed from the data. This left 49 participants. Amongst these, the average age was 28.4, with a standard deviation of 11.3. The youngest user was 18, and the oldest user was 74. The vast majority of participants were male; only 3 were female, and 1 user declined to provide a gender. Also, 35 of the users were undergraduate or graduate students, including 2 postdocs.

Almost all users had been programming in Java for at least 2 years, except one person who had written a Java program for a class, and one person who had no Java experience at all, but 20 years of general programming experience.

## 7.1 Setup

The survey consisted of a series of web forms that users could fill out from any web browser. Subjects were first asked to fill out a form consisting of demographic information, after which they were presented with a set of instructions, and a series of tasks.

### 7.1.1 Instructions

Each task repeated the instructions, as shown in Figure 12. Users were meant to associate the small icon next to the text field with the large icon at the bottom of the page. Next to the large icon were printed instructions. The instructions asked the user to infer what the program did at the location of the text field in the code, and to write an expression describing the proper behavior. The instructions also prohibited users from looking online for answers.

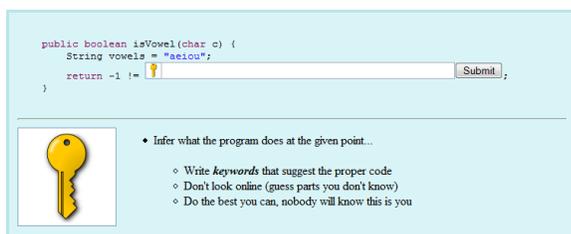
Different icons represented different variants of the instructions. There were three variants: *Java*, *pseudocode*, and *keywords*. The Java and pseudocode variants simply asked the user to "write Java code" or "write pseudocode" respectively. The keywords variant said "Write **keywords** that suggest the proper code." None of the instructions provided examples of what users should type, in order to obtain naturalistic responses.

Each user saw two instruction variants: either Java and pseudocode, or Java and keywords.

### 7.1.2 Tasks

The survey consisted of 15 tasks. Each task consisted of a Java method with an expression missing, which the user had to fill in using Java syntax, pseudocode, or a keyword query. The 15 missing expressions are shown in Table 3. Figure 12 shows the context provided for task 5.

The same 15 tasks were used for each user, but the order of the tasks was randomized. Five of the tasks requested Java syntax, and these five tasks were grouped



**Fig. 12** Example of a task used in the web user study (task 5 from Table 3). This example asks the user to enter keywords that suggest the missing expression, but other users were asked to enter Java code or pseudocode for this task.

**Table 3** Missing Expressions for Tasks

task	desired expression
1	message.replaceAll(space, comma)
2	new Integer(input)
3	list.remove(list.length() - 1)
4	fruits.contains(food)
5	vowels.indexOf(c)
6	numberNames.put(key, value)
7	Math.abs(x)
8	tokens.add(st.nextToken())
9	message.charAt(i)
10	System.out.println(f.getName())
11	buf.append(s)
12	lines.add(in.readLine())
13	log.println(message)
14	input.toLowerCase()
15	new BufferedReader(new FileReader(filename))

together either at the beginning or the end of the experiment. The remaining ten tasks requested either pseudocode or keywords.

### 7.1.3 Evaluation

Each user's response to each task was recorded, along with the instructions shown to the user for that task. Recall that if a user omitted any response, or supplied a garbage answer for any response, then we removed all the responses from that user from the data.

Tasks 1 and 3 were also removed from the data. Task 1 was removed because it is inherently ambiguous without taking word order into account, since `message`, `space`, and `comma` are all of type `String`. Task 3 was removed because it requires a literal ('1'), which was not handled by this version of our algorithm.

The remaining responses were provided as keyword queries to the algorithm in the context of each task. The model supplied to the algorithm was constructed from a Java source file containing all 15 tasks as separate methods. The resulting model had 2281 functions and 343 types, plus a few functions to model the local variables in each task, so it is comparable in complexity to the models used in the artificial corpus study (Figure 9).

**Table 4** Query counts and statistics for each instruction type.

	Java	pseudo	keywords
query count	209	216	212
average keywords per query	4.05	4.28	3.90
standard deviation	1.17	1.95	1.62
min/max keyword count	1—8	2—14	1—12
query uses Java syntax	98%	73%	45%

## 7.2 Results

Table 4 shows the number of queries for each instruction type, along with various statistics. Many queries used some form of Java syntax, even when the instructions called for pseudocode or keywords. This is understandable, because the participants were experienced Java programmers, and the tasks were posed in the context of a Java method. In Table 4, “uses Java syntax” means that the query could compile as a Java expression in some context.

When specifically asked to write Java code, users wrote syntactically and semantically correct code only 53% of the time, since users were asked not to use documentation, and most errors resulted from faulty memory of standard Java APIs. For instance, one user wrote `vowels.find(c)` instead of `vowels.indexOf(c)` for task 5, and another user wrote `Integer.abs(x)` instead of `Math.abs(x)` for task 7. Some errors resulted from faulty syntax, as in `new Integer.parseInt(input)` for task 2. The number 53% is used as a baseline benchmark for interpreting the results of the algorithm, since it gives a feel for how well the users understand the APIs used for the tasks.

Overall, the algorithm translated 59% of the queries to semantically correct Java code. Note that this statistic includes *all* the queries, even the queries when the user was asked to write Java code, since the user could enter syntactically invalid Java code, which may be corrected by the algorithm.

In fact, the algorithm improved the accuracy of Java queries alone from the baseline 53% to 71%. The accuracies for translating pseudocode and keyword queries were both 53%, which is encouraging, since it suggests that users of this algorithm can obtain the correct Java code by writing pseudocode or keywords as accurately as they can write the correct Java code themselves.

A breakdown of the accuracies for each task, and for each instruction type, are shown in Figure 13.

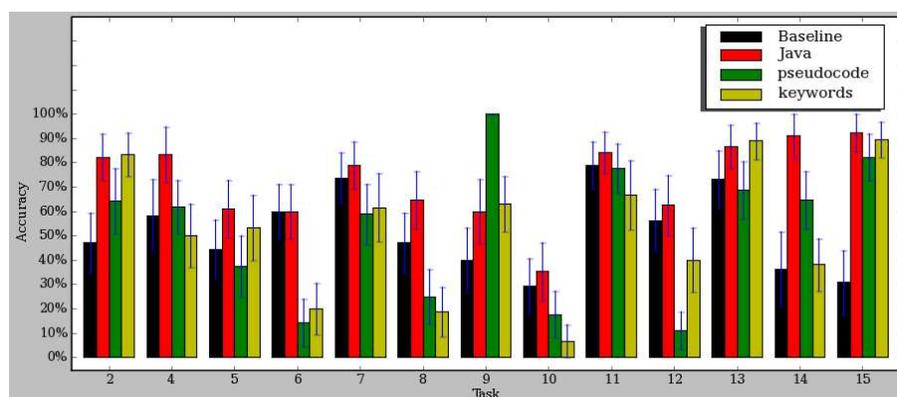
Tables 5 and 6 show a random sample of queries for each instruction variant that were translated correctly and incorrectly. The queries were quite varied, though it should be noted that many queries for pseudocode and keywords were written with Java style syntax.

## 8 Eclipse Plug-in

In order to get a feel for how keyword programming might actually be used in practice, we implemented it as an extension to Eclipse’s autocomplete mechanism, called Quack.<sup>1</sup>

A few of the items we had planned as future work seemed critical for achieving real-world usability, so we added these features to the plug-in. These included: returning a

<sup>1</sup> “Quack” is a very loose phonetic acronym for “Keyword Command Completion.”



**Fig. 13** Accuracy of the algorithm for each task, and for each instruction type, along with standard error. The “Baseline” refers to Java queries treated as Java, without running them through the algorithm.

**Table 5** Examples of queries translated correctly.

instructions	input	output	task
Java	Math.abs(x)	Math.abs(x)	7
	input.toInt()	Integer.parseInt(input.toString())	2
	tokens.add(st.nextToken())	tokens.add(st.nextToken())	8
pseudocode	letter at message[i]	message.charAt(i)	9
	System.out.println(f.name())	System.out.println(f.getName())	10
	input.parseInt()	Integer.parseInt(input)	2
keywords	vowels search c	vowels.indexOf(c)	5
	lines.add(in.readLine())	lines.add(in.readLine())	12
	buf.append(s);	buf.append(s)	11

**Table 6** Examples of queries translated incorrectly.

instructions	input	output	task
Java	return(x>=0?x:-x);	x.rotateRight(Integer.SIZE,...)	7
	tokens.append(st.nextToken())	st.nextToken(tokens.lastElem...)	8
	buf.add(s)	new Vector().add(buf)	11
pseudocode	(x < 0) ? -x : x	x.rotateLeft(Integer.SIZE,...)	7
	lines.append(in.getNext() );	lines.equals(in.getClass())	12
	input.lowercase();	input	14
keywords	Add s to buf	s.compareTo(buf.toString())	11
	in readline insert to lines	lines.contains(in.toString())	12
	print name of f	Integer.valueOf(f.getName())	10

ranked list of matching expressions instead of just one best-scoring expression; support for literals, such as numbers and quoted strings; and support for array indexing.

This section describes the changes made to the algorithm, followed by a discussion of the user interface, and the backend implementation of the plug-in. The following section presents a field test of the algorithm.

## 8.1 Algorithm Changes

We made three substantial changes to the algorithm: multiple query results, literals, and array indexing.

### 8.1.1 Multiple Query Results

Our original system uses a greedy algorithm to extract a function tree from the dynamic program. Unfortunately, it is unclear how to extend this algorithm to return multiple results. After trying several techniques, we settled upon using a different algorithm to extract a function tree from the dynamic program, namely A\* search.

The A\* algorithm is a heuristic search over a state space for the most valuable goal state, given a value function. (A\* is normally described as minimizing a cost function, but since we are trying to *maximize* the score of the function tree, we simply invert the usual description.) In our case, a state consists of a partially expanded function tree: some of the tree-nodes are functions, and some of the leaf-nodes are expansion points. Expansion points are represented by a desired return type, and a maximum height. Goal states are function trees with no expansion points.

Each state  $x$  is assigned a value  $f(x)$  which equals the known value  $g(x)$ , plus  $h(x)$ , which represents the possible additional value if all expansion points are expanded. More specifically,  $g(x)$  is the sum of the explanation vectors for all the function tree-nodes, and  $h(x)$  is the sum of the maximal explanation vectors for all the expansion points. In this case, a “maximal explanation vector” for an expansion point with desired type  $t$  and maximum height  $i$  is obtained by creating an explanation vector  $e_0, \dots, e_n$  where  $e_j$  is the max over the  $e_j$ 's for all the explanation vectors found in each  $bestRoots(t, i')$  where  $1 \leq i' \leq i$ .

Note that the heuristic we use for  $h(x)$  always overestimates the value of an expansion point. In terms of A\*, this is an admissible heuristic, which makes this use of A\* optimal for extracting a function tree from the dynamic program. However, the dynamic program may not hold the optimal function tree (since we only keep  $r$  functions in each cell), so the algorithm as a whole is not optimal. The running time for this algorithm also depends on the keyword query and the model, but empirical tests show that it is comparable to the greedy algorithm.

In the plug-in, we run A\* until we have obtained three complete function trees, or until we have processed 2000 elements of the priority queue. These three function trees are presented to the user as a list of query results. These numbers are arbitrary, and were chosen more because we expected that users would not want to look through more than 3 items, and not because the algorithm was not fast enough to produce more results.

### 8.1.2 Literals

The solution for supporting literals was far less drastic. We essentially added a new type of function to the model which had a regular expression as a label. For instance, integer literals were represented with the function (`int, ([-+]?[0-9]+)`).

Note that care had to be taken in our keyword query parser so that an input like `abs -7.2e2` would be tokenized into the keywords `abs`, and `-7.2e2`, as opposed to `abs`, `7`, `2`, `e`, `2`. We also needed to pay special attention to handle cases where the keyword query included Java style punctuation, as in `Door5.open()`. In this case, we want the

keywords **door**, **5** and **open**, and we specifically do not want the dot ‘.’ to be absorbed into the **5** to yield the keyword “**5.**”.

Recall from the discussion of explanation vectors in section 4.1 that a function (`int`, `([-+]?+)`) for the keyword query **add 333 777** will have the explanation vector:

$(e_0, 0_{\text{add}}, 0.5_{\text{333}}, 0.5_{\text{777}})$

Unfortunately, when it comes time to render the function tree as Java code, this does not tell us whether we should render a specific invocation of the function as **333** or **777**. To deal with this, we render the left-most literal function, with the left-most literal token (of the appropriate type) in the keyword query, and we render the next left-most literal function with the next left-most literal token, etc.

Note that this implementation requires users to explicitly delimit quoted strings. Previous keyword programming implementations such as Chickenfoot and Koala supported unquoted strings, but handled them as special cases. We believe strings may also need to be handled as a special case in this system, but this is left for future work.

### 8.1.3 Array Indexing

Our solution for array indexing was also pretty straightforward; for every array type in  $T$ , e.g. `Object[]`, we added a function to  $F$  of the form (`Object`, `()`, `Object[]`, `int`). Note that the label is blank, so this function can only be inferred to satisfy type constraints. A more appropriate choice may have been the label `()`; however, this would also require adjusting the query parser to treat `[` and `]` as keywords, as opposed to ignoring these characters as punctuation.

## 8.2 User Interface

Figure 14 shows a typical use of the plug-in. The user begins by entering a keyword query at the location they would like a Java expression to appear. In this example, they enter the keywords **add line**. Next, the user invokes Eclipse’s autocomplete mechanism, which is usually mapped to Ctrl-Space. This brings up a popup menu with alternatives for completing the current query. Quack labels its results in the menu to distinguish them from items generated by the existing autocomplete mechanism, which may also be found in the menu. Pressing Enter selects the top item in the list, in this case replacing the keyword query with the expression `array.add(src.readLine())`.

Although the illustration shows the keyword query on a blank line, the query may also be entered inside a larger expression. To find the start of the keyword query, the plug-in considers the text on the current line preceding the cursor, and finds the longest suffix that contains only keyword tokens. Tokens include literals like **5.0**, but not punctuation like the dot separating an object from a method call in `out.print()`. We also make a special case for “return  $x$ ” and “throw  $x$ ”, since Java syntax does not require any punctuation between these words and the expression  $x$ . Since our algorithm does not generate return or throw statements itself, we do not allow the query to begin with either of these words.

As an example of finding the keyword query, consider the following line:

```
v.add(f name|)
```



Fig. 14 An illustration of the use of the Quack Eclipse Plug-in.

The | represents the cursor location where the user presses Ctrl-Space. In this case, the longest suffix preceding the cursor is **f name**. The next longest suffix is '(f name)', which is rejected because the '(' is not a valid keyword query token.

If the keyword query is a single keyword, we treat this as a special case. Consider the following example:

```
v.add(f.getNam|
```

When the user invokes autocomplete, the Quack plug-in will find the keyword query **getNam**, since it is the longest valid suffix preceding the cursor. However, **Nam** is not complete keyword in the desired function, so Quack will fail to find the correct function. One way to fix this would be to make Quack support prefix matching of function names, but for now, we prevent the plug-in from making suggestions in this case. We feel this is a good idea in early stages of the prototype anyway, since it prevents Quack from being invoked in many of the cases where the user would expect Eclipse's native autocomplete to function. Whereas, if the user enters a query like **f**

**name**, then Eclipse's native autocomplete is unlikely to yield any results, since **f name** is syntactically invalid.

If the user wants to enter a query consisting of a single keyword, they must enter a space after the query, as in:

```
v.add(name |
```

One problem with this approach is that it suggests to the user that they could use Quack to complete the latter part of an expression with a receiver object; for instance, if they typed:

```
v.add |
```

In this case, the keyword query is **add**, but the plug-in does not know about the implicit parameter **v** which needs to be passed to **add**. Supporting this is future work.

As a fallback mechanism, the user may select the entire query, and then invoke Control-Space. If the Quack plug-in sees that the user has selected some text, then it will always assume it is a keyword query. This maintains a separation with Eclipse's native autocomplete, since users generally do not select text before invoking native autocomplete.

### 8.3 Backend

When the user invokes Quack, the plug-in finds the keyword query as described in the previous section, and then obtains the remaining parameters needed for the search algorithm: the model and the desired return type, if any. This section describes how these parameters are obtained.

#### 8.3.1 Creating the Model

First, the plug-in creates a model  $M$  based on all the classes which are visible in the current source file. A class is visible if there are any identifiers bound to it, or any method calls which return it. The plug-in also recursively processes classes which are returned by methods in the classes it has found so far, up to a certain depth, which is determined by the height of the dynamic program used by the algorithm, see section 4.

The plug-in keeps track of the model used by the most recent completion, and reuses this model if the user invokes Quack again in the same Java source file. Other options might be caching model information for each package, or even the entire project. The issue here is that different functions are visible to different classes, and we did not engineer a way of storing this information in the model. This is a possibility for future work.

#### 8.3.2 Add Local Variables and Member Fields and Methods

In the previous step, the plug-in builds a model for the Java functions visible from the current source file, but it needs to do another pass to find the local variables, member fields and methods that are visible from the particular scope of the keyword query. This information is not cached, since it is not the same for every location within the same source file.

### 8.3.3 Determine Return Type

Next, the plug-in tries to determine what Java types are valid for an expression at the location of the keyword query. In the example from Figure 14, the keyword query is on a blank line, and so any return type is valid. This is represented by making the desired return type `void`, which we implement as a super type for all other types.

When the keyword query appears inside a larger expression or statement, the plug-in may be able to infer more about the return type. Consider:

```
String a = f name|
```

When the user invokes autocomplete in this case, the Quack plug-in uses `f name` as the keyword query, and infers a return type of `java.lang.String`, since the keyword query is sitting on the right-hand-side of an assignment to a String.

### 8.3.4 Generate Results

Finally, the plug-in uses the keyword query, model and desired return type, and invokes the algorithm. The algorithm returns up to three function trees, in rank order, rendered as Java expressions. These expressions are added to the autocomplete menu, with their relevance value set so that they always appear at the very top.

## 9 Preliminary Field Study

In order to get a feel for the usefulness and accuracy of Quack in realworld programming scenarios, we conducted a preliminary field study, giving the plug-in to three developers in our research lab (two PhD students, and one postdoc). All three were working on Java projects for their research, and agreed to use the plugin regularly for a week. They were each paid \$50 for their time.

We trained each participant for about 30 minutes at the beginning of the study. We also interviewed each participant at the end of the study. See section 10 for a discussion of these interviews.

The plug-in recorded statistics about the use of the plug-in by each participant. Whenever the user invoked autocomplete on a keyword query, the plug-in recorded the keyword query, along with the expressions that were displayed to the user. If the user selected one of the proposed expressions, the plug-in recorded which option they selected. Note that users might select an item from the list, even if it was not quite what they wanted (one user specifically reported doing so). The plug-in did not record enough information to determine whether the user kept the completion they selected.

### 9.1 Quantitative Results

Quack recorded 160 keyword query log entries for all the users. We manually grouped these into 90 sessions, where each session involved one or more queries aimed at generating a particular expression in a particular location in the code. An example session is shown in Figure 15. In this session, the user began by entering the keyword query `names put`, and invoking Quack. However, the results did not contain the desired

---

```

excerpt from log  query: names puts

result: namesToValues.putAll(namesToValues)
result: new Properties().putAll(namesToValues)
result: System.getProperties().putAll(namesToValues)

query: names put variable vt

result: namesToValues.put(variableName, vt)
result: equals(namesToValues.put(variableName, vt))
result: namesToValues.put(variableName, vt).vals

selected: namesToValues.put(variableName, vt)

```

---

**Fig. 15** One session of keyword query invocations taken from the field study. The user selected nothing on the first attempt, altered their query, and then selected the top result.

**Table 7** Session counts for each user; the number in parentheses is the average number of queries for each session.

	User A	User B	User C
Ended with Selection	10 (1.3)	23 (1.5)	17 (1.25*)
Ended with No Selection	6 (1.3)	11 (1.3)	12 (3.4)
Error	1 (4)	1 (1)	0
Method Declaration	0	1 (1)	2 (1)
Constructor Completion	0	5 (1.2)	1 (3)

\* Excludes an outlier where the user tried 11 times before succeeding.

expression, so they added the keywords **variable** and **vt** to their query, and invoked Quack again. This time, the user selected the top result returned by Quack.

Sessions were classified by whether the user eventually selected a result from Quack, except for a few special cases. These exceptions were cases where the plug-in either had an error, or where the user probably did not intend to invoke Quack (note that Quack is tied to Eclipse’s autocomplete system, so the user may have been intending to use some other feature of autocomplete). These error classes are enumerated below:

1. Error: Sessions in which the plug-in threw an exception, and therefore didn’t display any results.
2. Method Declaration: Session where the user entered keywords that appear to be part of a method declaration, like **public static void createPC**. Quack does not support method declarations, and the plug-in query detector should filter these cases, but didn’t due to a bug.
3. Constructor Completion: Session where the user appears to have entered a prefix to a constructor call, as in **new Wek**. In these cases, the user probably intended to call upon Eclipse’s regular autocomplete to complete the name of the class, rather than Quack.

Table 7 shows the number of sessions invoked by each user, grouped into the categories above. If we ignore the special error classes, then selection rates ranged from 58% to 68%. Most selections were obtained on the first invocation of Quack.

To put these selection rates in perspective, it is useful to consider how many keywords each user provided compared to the number of keywords in the expressions they selected. These numbers are provided in table 8.

We can see an interesting difference in the usage pattern of user A compared with the usage pattern of users B and C. User A tended to provide more keywords, and

**Table 8** Keyword counts for queries where the user selected one of the results, along with the number of keywords in the results.

	User A	User B	User C
Average number of keywords in query	3.5	2.3	2.6
Average number of keywords in result	4.2	4.8	5.1

seemed to generate shorter expressions. In fact, this user would sometimes provide all the keywords, and essentially use Quack to fill in the punctuation. For example, they converted `system out println` into `System.out.println()`. Sometimes they would use all the keywords, but in a different order, as in: `a1 getCond takeDiscreteFacts` to get `takeDiscreteFacts(a1.getCond())`.

On the other hand, both users B and C tended to use Quack to infer a large expression from a small number of keywords. For instance, converting `DOCUMENTS` into `ChronKB.CHRON_ONT_DOCUMENTS`. These usage patterns and others are discussed in more detail in the next section.

## 10 Discussion

This section discusses some of the advantages and limitations of the keyword programming technique, based on the user studies described earlier as well as the authors' personal experience using the plug-in. We identify some common usage patterns from empirical observation, and we suggest some potential improvements to the algorithm based on common modes of failure.

### 10.1 Type Constraints

One field study user said that the plug-in tended to work well in cases where there was a lot of context type information; for instance, when completing a keyword query which was an argument to a method. In this case, the plug-in could often infer the return type based on the method signature.

Another user mentioned that the plug-in was useful in cases where they had forgotten a method name completely, and the plug-in had some chance of reminding them of the name given other constraints, like arguments to the method.

The third user mentioned that the plug-in worked well in smaller classes, presumably because there was less ambiguity.

### 10.2 Learning Shortcuts

Similar to how Eclipse allows users to type `sysout` and then Control-Space to produce `System.out.println()`, Quack effectively supports a variety of shortcuts for other common commands. One idiom we commonly use is `exit 0`, which produces `System.exit(0)`. It is also possible to type `out "hello"` to produce `System.out.println("hello");` one user used this idiom a couple of times.

### 10.3 Using Unique Keywords

When searching the Web for a particular page, it's often useful to think of an unusual keyword that's likely to appear on the target page. The same technique works in Quack. An example is the word **millis**, which is likely to produce `System.currentTimeMillis()`, since the word **millis** is so unlikely to appear in any other function. An example of this pattern from the user study is the conversion of **Address statistics** into `Pointer-AddressRanges.AddressInfo.dumpStatistics()`.

### 10.4 Avoiding Punctuation and Capitalization

Another common use case is simply typing a simple expression without punctuation, and letting Quack fill in the punctuation. A simple example from the field study is converting **it2 next** into `it2.next()`. We mentioned before that this was a common idiom for one of the field study participants. They would even go one step further, letting Quack combine keywords into camel-case identifiers, as exemplified by converting **f1Vector element at j** into `f1Vector.elementAt(j)`.

### 10.5 Unfamiliar APIs

Turning now to limitations, one field study user was doing work on an unfamiliar API, and they found that the plug-in was less useful in this setting. They found that they could not come up with keywords, since they did not know which functions to call. Some of the confusion stems from synonyms like “put” versus “add”, but some of the confusion stems from not understanding the model for how the API works.

One of the goals of keyword programming is to manage large APIs, even unfamiliar ones. An important area of future work will be to determine whether the accuracy of the current system is too low, or whether there is a fundamental reason why keyword programming will not work on unfamiliar APIs.

### 10.6 Misdetected Queries

Field study users reported that the mechanism for determining the start and end of the keyword query was sometimes confusing. One user specifically mentioned: “I forget all the symbols at the start or end that Quack looks at”. They provided the example of entering a query which included `set<string>`, and Quack would only consider the part of the query after the ‘>’.

A related problem has to do with how a keyword query is repaired. For instance, after viewing Quack's suggestions, a user may notice an error in the beginning of their keyword query. Unfortunately, after they repair this error, they must remember to bring their cursor back to the end of the entire query before invoking Quack again.

Errors of both sorts may be mitigated by having the user invoke Quack *before* entering their expression, and there may be other reasons for this as well, discussed below.

---

## 10.7 Understanding What Went Wrong

When Quack did not suggest the desired expression, it could be difficult to understand why. One reason Quack may not work is that a desired method call is a member of an unimported class. Another more subtle reason mentioned by a user is that certain fields may be invisible from the current scope, and the user doesn't realize that they are invisible. One common example of this is trying to access a non-static field from a static method.

These issues are problems with Eclipse's own autocomplete mechanism as well, but they are more difficult in Quack, because the user enters an entire query before they get feedback saying it doesn't work. Autocomplete tends to work one token at a time, so the user knows exactly which token Eclipse is having trouble finding.

A third source of errors comes from misspelled keywords. These can be difficult to detect amidst other sources of errors, though this particular problem is probably best addressed by adding spell correction to the system, which we discuss below.

## 10.8 Recognizing the Right Expression

One user reported a bad experience when Quack suggested something that was close to correct, but wasn't quite correct. They selected the option, not realizing that it was the wrong choice. This can happen with regular autocomplete as well, but the problem is aggravated in Quack for a couple of reasons. First, the user needs to recognize an entire expression, which can be difficult if the API is unfamiliar. Second, the user interface does not provide any documentation for the functions involved in the expression. Such documentation could be displayed in a popup next to the list of suggestions, as is already done for Eclipse's autocomplete suggestions.

## 10.9 Continuous Feedback

Another user suggested a more drastic change to the interface involving continuous feedback, and the ability to confirm certain hypotheses made by the system. For instance, the system might report that a likely binding for a particular keyword is to a particular local variable, and if the user could confirm this, then it may reduce the search space.

The combination of invoking Quack before entering a keyword query, and supplying continuous feedback, is an appealing direction to explore in future work.

## 10.10 A Priori Weights for Keywords

In the web user study, the algorithm incorrectly translated **print name of f** to `Integer.valueOf(f.getName())`. (The correct expression should have been `System.out.println(f.getName())`.) Since the algorithm could not find an expression that explained all the keywords, it settled for explaining **name**, **of**, and **f**, and leaving **print** unexplained. However, **print** is clearly more important to explain than **of**.

One possible solution to this problem is to give a priori weight to each word in an expression. This weight could be inversely proportional to the frequency of each word in a corpus. It may also be sufficient to give stop words like **of**, **the**, and **to** less weight.

### 10.11 A Priori Weights for Functions

The query `println f name` in task 10 of the web user study was translated to `System.err.println(f.getName())`. A better translation for this task would be `System.out.println(f.getName())`, but the algorithm currently has no reason to choose `System.out` over `System.err`. One way to fix this would be to have a priori function weights. These weights could also be derived from usage frequencies over a corpus.

Of course, these function weights would need to be carefully balanced against the cost of inferring a function. For instance, the input `print f.name` in task 10 was translated to `new PrintWriter(f.getName())`, which explains all the keywords, and doesn't need to infer any functions. In order for the algorithm to choose `System.out.println(f.getName())`, the cost of inferring `System.out`, plus the weight of `print` as an explanation for the keyword `print` would need to exceed the weight of `new PrintWriter` as an explanation for `print`.

### 10.12 Spell Correction

A user in the field study mentioned the need for spell correction. This problem could also be seen in the web user study. Many users included `lowercase` in their queries for task 14. Unfortunately, the algorithm does not see a token break between `lower` and `case`, and so it does not match these tokens with the same words in the desired function `toLowerCase`. One solution to this problem may be to provide spell correction, similar to [5]. That is, a spell corrector would contain `toLowerCase` as a word in its dictionary, and hopefully `lowercase` would be corrected to `toLowerCase`.

### 10.13 Synonyms

Another frequent problem involved users typing synonyms for function names, rather than the actual function names. For instance, many users entered `append` instead of `add` for task 8 of the web user study, e.g., `tokens.append(st.nextToken())`. This is not surprising for programmers who use a variety of different languages and APIs, in which similar functions are described by synonymous (but not identical) names.

An obvious thing to try would be adding `append` to the label of the function `add`, or more generally, adding a list of synonyms to the label of each function. To get a feel for how well this would work, we ran an experiment in which each function's label was expanded with all possible synonyms found in WordNet [1]. This improved some of the translations, but at the same time introduced ambiguities in other translations.

An example of where the synonyms helped was in translating inputs like `tokens.append(st.nextToken)` for task 8, into `tokens.add(st.nextToken())`. An example of where the synonyms caused problems was task 6, where the input `numberNames.put(key,value)` was translated erroneously as `String.valueOf(numberNames.keySet())`. The reason for this translation is that `put` is a synonym of `set` in WordNet, and so the single function `keySet` explains both the keywords `put` and `key`. Also note that the algorithm uses a heuristic which prefers fewer function calls, and since `String.valueOf(numberNames.keySet())` has only three function calls (where `String.valueOf` is a single static function), this interpretation is favored over `numberNames.put(key, value)` which has four function calls.

Overall, the accuracy decreased slightly from 59% to 58%. It may be more effective to create a customized thesaurus for keyword programming, perhaps by mining the documentation of programming languages and APIs for the words that programmers actually use to talk about them, but this remains future work.

In general, it is difficult to anticipate the repercussions of changing the heuristics in the algorithm. Developing a principled manner for adjusting these heuristics is another important area for future work.

## 11 Future Work

The previous section discussed a number of areas for future work in the user interface and underlying algorithm. Another ripe area for improvement lies in the model, and what Java expressions it can represent.

### 11.1 Model Extensions

Our current model of Java is fairly limited. We expanded it slightly in the Eclipse Plug-in (section 8) to include literals as well as array indexing. One user from the field study specifically called for the plug-in to generate Java statements that it currently doesn't support. In this section, we describe some of the extensions that we believe would be useful, with some thoughts about how they could be represented in the current system, or how the system could be expanded to support them.

#### 11.1.1 Operators

Java operators can map to functions in the natural manner, but we require multiple versions of them to support all the primitive types that use them. For example, we require (`int`, `+`), (`int`, `int`) separate from (`long`, `+`), (`long`, `long`). It might seem like we could just have (`double`, `+`), (`double`, `double`), since all the other numeric primitives are subtypes of `double`. However, this wouldn't allow us to add two numbers, and pass the result to a function that requires an `int`.

#### 11.1.2 Assignment

One user proposed enhancing Quack to support variable declaration statements by typing something like `vector of fact v`, and having Quack generate `Vector<Fact> v = new Vector<Fact>()`. Assignment is more complicated than other operators, however. Say we want to allow the assignment `x = y`, where `x` is an `int` and `y` is a `short`. We could add (`int`, `(=)`), (`int`, `int`). Unfortunately, this doesn't prevent us from passing subtypes of `int` to the left-hand side of the assignment, so this wouldn't prevent `y = x`. It also wouldn't prevent `5 = x`.

One approach we experimented with is adding a special set-function for each variable. In this example, we would add (`int`, `(x =)`), (`int`). Note that the function name includes the `=` symbol. This seems to work, but it does require adding lots of new functions.

A cleaner solution might involve adding more types; in particular, an "assignable" or "reference" type. So the variable `int x` would be accessible with the function

(`ref:int`, (`x`)). Then we would have a function for integer assignment: (`int`, (`=`), `ref:int`, `int`). We would also make `ref:int` a subtype of `int`, so that we could still use `x` on the right-hand-side of an assignment. This technique still requires an assignment function for each type, but not for each variable.

### 11.1.3 Other

A general keyword programming system should include the full range of Java constructs, including control flow statements and class declarations. One possible paradigm would use line-by-line completions of these constructs. For instance, if the user entered the keywords `if x == y`, the system might suggest “`if (x == y) {`”, or create a template of an `if` statement with `x == y` filled in. The function for `if` would then look like (`void`, (`if`), `boolean`). Note that this function is not a function in the traditional sense of a callable procedure; rather, it is a function that generates code for the user to further edit. This is fine for our system since it is a code completer, and not an actual interpreter.

## 12 Related Work

This work builds on our earlier efforts to use keywords for scripting – i.e., where each command in a script program is represented by a set of keywords. This approach was used in Chickenfoot [5] and Koala [4]. The algorithms used in those systems were also capable of translating a sequence of keywords into function calls over some API, but the APIs used were very small, on the order of 20 functions. Koala’s algorithm actually enumerates all the possible function trees, and then matches them to the entire input sequence (as opposed to the method used in Chickenfoot, which tries to build trees out of the input sequence). This naive approach only works when the number of possible function trees is extremely small, which was true for Chickenfoot and Koala because they operate on web pages. Compared to Chickenfoot and Koala, the novel contribution of the current paper is the application of this technique to Java, a general purpose programming language with many more possible functions, making the algorithmic problem more difficult.

This work is also related to work on searching for examples in a large corpus of existing code. This work can be distinguished by the kind of *query* provided by the user. For example, Prospector [6] takes two Java types as input, and returns snippets of code that convert from one type to the other. Prospector is most useful when the creation of a particular type from another type is non-obvious (i.e. you can’t simply pass it to the constructor, and other initialization steps may be involved). Another system, XSnippet [9], retrieves snippets based on context, e.g., all the available types from local variables. However, the query result is still a snippet of code that achieves a given type, and the intent is still for large systems where the creation of certain types is nontrivial. A third approach, automatic method completion [2], uses a partially-implemented method body to search for snippets of code that could complete that method body.

The key differences between our approach and these other systems are:

1. The user’s input is not restricted to a type, although it *is* constrained by types available in the local context. Also, the output code may be arbitrary, not just

---

code to obtain an object of a certain type. For instance, you could use a keyword query to enter code on a blank line, where there is no restriction on the return type.

2. Our approach uses a guided search based on keywords provided by the user. These keywords can match methods, variables and fields that may be used in the expression.
3. Our approach generates new code, and does not require a corpus of existing code to mine for snippets. In particular, users could benefit from our system in very small projects that they are just starting.

There is also substantial work on searching for reusable code in software repositories using various kinds of queries provided by the user. Both [8] and [13] explore the use of method type signatures as a query mechanism, including the idea of matching relaxed (inexact) versions of method signatures. The use of formal specifications as a search query is explored in [3] and [14], including matching behavioral subtypes of a specification in [14]. A faceted classification scheme for component retrieval is proposed in [7]. This approach requires meta-data about components and functions, but this information may be useful in our algorithm as well, particularly a list of synonyms for function names. Other work in the area of software reuse includes a hybrid query system which takes advantage of various other approaches [10], and systems which actively search for reusable code as the programmer is working, rather than requiring the user to explicitly invoke a query [11,12]. These systems are aimed at the problem of identifying and selecting *components* to reuse to solve a programming problem. Our system, on the other hand, is aimed at the coding task itself, and seeks to streamline the generation of correct code that *uses* already-selected components.

### 13 Conclusion

We have presented a novel technique for *keyword programming* in Java, where the user provides a keyword query and the system generates type-correct code that matches those keywords. We presented a model for the space over which the keyword search is done, and gave an efficient search algorithm. Using example queries automatically generated from a corpus of open-source software, we found that the type constraints of Java ensure that a small number of keywords is often sufficient to generate the correct method calls.

We also solicited keyword queries from users in a web based survey, and found that the algorithm could translate keyword queries with the same accuracy as users could write unassisted Java code themselves. We also identified several classes of errors made by the algorithm, and suggested possible improvements.

Finally, we created a user interface for the algorithm in the form of an plug-in that extends the autocomplete feature of Eclipse. In creating this plug-in, we explored a couple of extensions to the algorithm, and we conducted a field test of its usefulness in practice.

The long-term goal for this work is to simplify the usability barriers of programming, such as forming the correct syntax and naming code elements precisely. Reducing these barriers will allow novice programmers to learn more easily, experts to transition between different languages and different APIs more adroitly, and all programmers to write code more productively.

**Acknowledgements** This work was supported in part by the National Science Foundation under award number IIS-0447800, and by Quanta Computer as part of the TParty project. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

## References

1. C. Fellbaum, editor. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.
2. R. Hill and J. Rideout. Automatic Method Completion. *Proceedings of Automated Software Engineering (ASE 2004)*, pp. 228–235.
3. J.-J. Jeng and B. H. C. Cheng. Specification Matching for Software Reuse: A Foundation. In *Proceedings of the 1995 Symposium on Software Reusability*, pp. 97–105, 1995.
4. G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. *Proceedings of CHI 2007*, to appear.
5. G. Little and R. C. Miller. Translating Keyword Commands into Executable Code. *Proceedings of User Interface Software & Technology (UIST 2006)*, pp. 135–144.
6. D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 48–61.
7. R. Prieto-Diaz and P. Freeman. Classifying Software for Reusability. *IEEE Software*, 4(1):6–16, 1987.
8. M. Rittri. Retrieving library identifiers via equational matching of types. *Proceedings of the tenth international conference on Automated deduction*, pp. 603–617, 1990.
9. N. Sahavechaphan and K. Claypool. XSnippet: Mining For Sample Code. *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA 2006)*, pp. 413–430.
10. N. Tansalarak and K. T. Claypool. Finding a Needle in the Haystack: A Technique for Ranking Matches between Components. In *Proceedings of the 8th International SIGSOFT Symposium on Component-based Software Engineering (CBSE 2005): Software Components at Work*, May 2005.
11. Y. Ye, G. Fischer, and B. Reeves. Integrating active information delivery and reuse repository systems. In *International Symposium on Foundations of Software Engineering*, pp. 60–68, November 2000.
12. Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pp. 513–523, May 2002.
13. A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, April 1995.
14. A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.