Keyword Programming in Java

Greg Little and Robert C. Miller
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139
{glittle,rcm}@mit.edu

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 keywords provided by the user into a valid expression. Prior work has demonstrated the feasibility and merit of this approach in limited domains. This paper presents a new algorithm that scales to the much larger domain of general-purpose Java programming. We tested the algorithm by extracting keywords from method calls in open source projects, and found that it could accurately reconstruct over 90% of the original expressions. We also conducted a study using keywords generated by users, whose results suggest that users can obtain correct Java code using keyword queries as accurately as they can write the correct Java code themselves.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques–program editors, object-oriented programming

General Terms

Experimentation, Languages

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 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.

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.

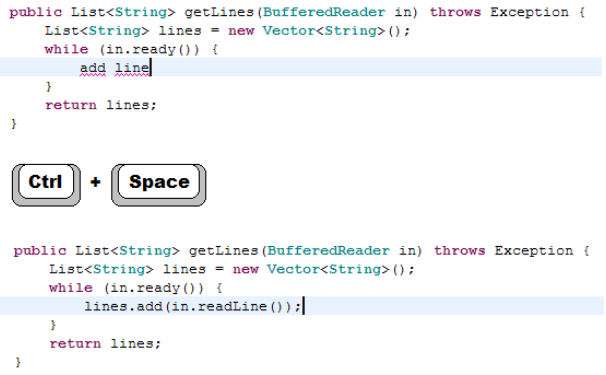

Figure 1: In keyword programming, the user types some keywords, presses a completion command (such as Ctrl-Space in Eclipse), and the keywords are translated into a valid expression.

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).

In the next section, we present a model, which is followed by a problem statement. We then present the algorithm, and the two evaluations. 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 anything 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 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 a method name, we break up the name at 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.

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 $T \times L \times T \times ... \times T$. 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`, (**to**, **string**), `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.

In particular, a node is 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., $|children(n)| = |params(func(n))|$. We also require that the return types from the children fit into the parameters, i.e.,
$\forall_i ret(func(children(n)_i)) \in sub(params(func(n))_i)$.

Note that in the end, the system renders the function tree as a syntactically-correct and type-correct expression in the underlying language.

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 Methods

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

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

Static methods do not need a receiver object—it is optional. To support the optional argument, 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

We support generics 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, we include 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, although we have not yet done a formal evaluation of them. These include numeric and string literals, variable assignment, and array indexing. We have also considered ways to model control flow. 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 can be regarded as a dynamic program where we fill out a table of the form $func(t, i)$, which tells us which function with return type t is at the root of a function tree that provides the best match for the keywords, assuming the function tree can be at most height i. The table also records the *score*, the degree of match of this function tree with the keywords.

Calculating $func(t, 1)$ for all $t \in T$ is relatively easy. We only need to consider functions that take no parameters, since our tree height is bounded by 1. For each such function f, we give it a score based on its match to the user's keywords, and we associate this score with $ret(f)$. Then for each $t \in T$, we update $func(t, 1)$ with the best score associated with any subtype of t.

Instead of a scalar value for the score, we use an *explanation vector*. We will explain what this is before talking about the next iteration of the dynamic program.

4.1 Explanation Vector

The idea of the explanation vector is to encode how well we have explained the input keywords. If we have n keywords $k_1, k_2, ..., k_n$, then the explanation vector has $n + 1$ elements $e_0, e_1, e_2, ..., e_n$. 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 ensure that the resulting elements $e_1, e_2, ..., e_n$ are capped at 1, since the most we can explain a particular keyword is 1.

Explanation vectors are compared by summing each vector's elements to produce a scalar score, and then comparing those scores.

Before we do anything else, 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:

$(e_0, 1_{\mathbf{is}}, 0_{\mathbf{queue}}, 1_{\mathbf{empty}})$

Unmatched keywords are penalized by subtracting 0.01 from e_0 for each word appearing in either the input or $label(f)$, but not both. In this case, e_0 is -0.01, since the word **queue** does not appear in $label(f)$.

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:

$(e_0, 0.5_{\textbf{node}}, 0_{\textbf{parent}}, 0_{\textbf{remove}}, 0.5_{\textbf{node}})$

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.

In this case we set e_0 to -0.03, since there are three words that appear in the input, but not in the function label (we include one of the **node** keywords in this count, since it only appears once in the label).

4.2 Next Iteration

In subsequent iterations of the dynamic program, the goal is to compute $func(t, i)$ for all $t \in T$, given the elements of the table that have already been computed, i.e., $func(t', j)$ for all $t' \in T$ and $j < i$. The basic idea is to consider each function f, and calculate an explanation vector by summing the explanation vector for f itself, plus the explanation vector for each parameter type p found in $func(p, i-1)$.

We can do this, but there is a problem. We no longer know that we have the optimal explanation vector possible for this function at this height; consider the following input:

add x y

and assume the model contains three functions:

(`int`, (**add**), `int`, `int`)
(`int`, (**x**))
(`int`, (**y**))

If we look in $func($`int`$, 1)$, we will see either (`int`, (**x**)), or (`int`, (**y**)). Let's assume it is (`int`, (**x**)). Now consider what happens in the next iteration when we are processing the function (`int`, (**add**), `int`, `int`). We take the explanation vector (-0.02, $1_{\textbf{add}}$, $0_{\textbf{x}}$, $0_{\textbf{y}}$), and we add the explanation vector found in $func($`int`$, 1)$, which is (-0.02, $0_{\textbf{add}}$, $1_{\textbf{x}}$, $0_{\textbf{y}}$). This gives us (-0.04, $1_{\textbf{add}}$, $1_{\textbf{x}}$, $0_{\textbf{y}}$).

Now we want to add the explanation vector for the second parameter, which is also type `int`. We look in $func($`int`$, 1)$ and find (-0.02, $0_{\textbf{add}}$, $1_{\textbf{x}}$, $0_{\textbf{y}}$) again. When we add it, we get (-0.06, $1_{\textbf{add}}$, $1_{\textbf{x}}$, $0_{\textbf{y}}$), since the keyword components are capped at 1.

But what if we had found the explanation vector for (`int`, (**y**))? Then we could have gotten (-0.06, $1_{\textbf{add}}$, $1_{\textbf{x}}$, $1_{\textbf{y}}$), which is better.

To get around this problem, we store the top r functions at each $func(t, i)$, where r is an arbitrary constant. In our experiments, we chose $r = 3$, except in the case of $func($`java.lang.Object`$, i)$, where we keep the top 5 (since many functions return this type).

Now when we are considering function f at height i, and we are adding explanation vectors for the parameters, we are greedy: we add the explanation vector that increases our final vector the most, and then we move on to the next parameter. Note that if our parameter type is p, we consider all the explanation vectors in each $func(p, j)$ where $j < i$.

```
procedure EXTRACT TREE(t, h, e)
 for each f ∈ func(t, i) where i ≤ h
   do { /* create tuple for function tree node */
        n ← (f)
        e_n ← e + expl(f)
        /* put most specific types first */
        P ← SORT(params(f))
        for each p ∈ P
          do { n_p, e_p ← EXTRACT TREE(p, i − 1, e_n)
               /* add n_p as a child of n */
               n ← append(n, n_p)
        if e_n > best_e
          then { best_e ← e_n
                 best_n ← n
 return (best_n, best_e)
```

Figure 2: Pseudocode to extract a function tree.

4.3 Extraction

After we have run the dynamic program to some arbitrary height h (in our case, $h = 3$), we need to extract a function tree.

We use a greedy recursive algorithm (see Figure 2) 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. Note that we sort the parameters such that the most specific types appear first (t_1 is more specific than t_2 if $|sub(t_1)| < |sub(t_2)|$).

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. Now if we assume that every function takes p parameters, every type has s subtypes, and every type is returned by f functions; then it takes $O(h(|F|phr + |T|sf))$ time to fill out the table. Extracting the best function tree requires an additional $O((hrp)^h)$ time, assuming we know the return type; otherwise it takes $O(|T|(hrp)^h)$ time.

In practice, the algorithm is able to generate 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.

Project	Class Files	LOC	Test Sites
Azureus	2277	339628	82006
Buddi	128	27503	7807
CAROL	138	18343	2478
Dnsjava	123	17485	2900
Jakarta CC	41	10082	1806
jEdit	435	124667	25875
jMemorize	95	14771	2604
Jmol	281	88098	44478
JRuby	427	72030	19198
Radeox	179	10076	1304
RSSOwl	201	71097	23685
Sphinx	268	67338	13217
TV-Browser	760	119518	29255
Zimbra	1373	256472	76954

Table 1: Project Statistics

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.

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.

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. We also report the number of possible test sites, which we discuss 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 inner classes since it simplifies our automated testing framework. Finally, we discard test sites with only one keyword as trivial.

```
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);
    }
    ⋮
}
```

Figure 3: Example Test Site

Figure 3 shows a valid test site highlighted 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 3 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 include other information about the test, including:

- **# Keywords:** the number of keywords in the keyword query.
- **time:** how many seconds the algorithm spent searching for a function tree. This does not include the time taken to construct the model. 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.
- $|T|$**:** the number of types in the model constructed at this test site.
- $|F|$**:** the number of functions in the model constructed at this test site.

# 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

Table 2: Samples given # Keywords

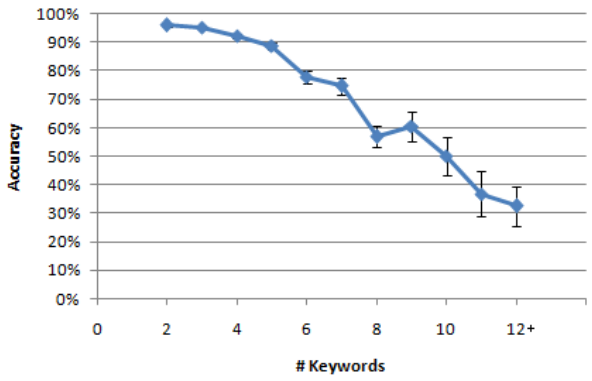

Figure 4: Accuracy given # keywords. Error bars show standard error.

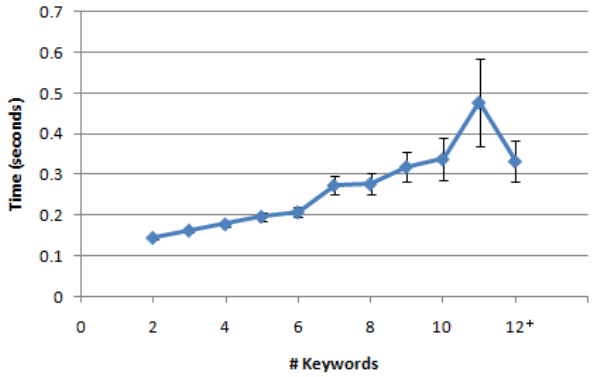

Figure 5: Time given # keywords. Error bars show standard error.

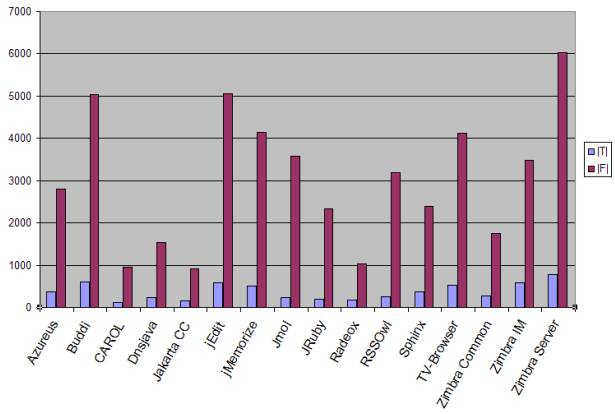

Figure 6: Average size of T and F for different projects.

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 500 from each of them). This gives us 8000 test sites. For each test site, we ran the algorithm once as 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 4 shows the accuracy of the algorithm given a number of keywords. The overall accuracy is 91.2%, though this is heavily weighted to inputs with fewer keywords, based on our sample sizes.

Figure 5 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 6 shows the average size of T and F for each project. The average size of F tends to be much larger than T. Figure 7 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 14000 functions.

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, 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 8 shows the number of keywords we provided as input. The table shows the accuracy for different expression lengths (measured in keywords).

6.4 Discussion

Our goal in running these tests was to determine whether keyword programming could be done in Java, or if the search space was simply too big. The results suggest that the problem is tractable: a simple algorithm can achieve a modest speed and accuracy.

The speed is reasonable for an Eclipse autocomplete-style plug-in; most queries are resolved in less than 500 milliseconds. Note that we didn't include the time it takes to build the model in these measurements, since the model can be

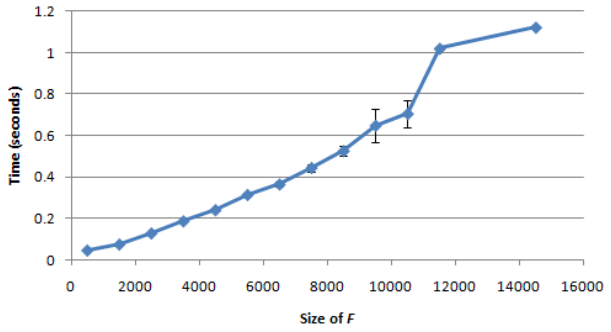

Figure 7: Time given size of F. Error bars show standard error.

Keywords provided \ 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

Figure 8: Accuracy of inference (1 is 100%)

constructed in the background, before the user submits a query. Constructing the model from scratch can take a second or more, but a clever plug-in 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. We also explore possible improvements to the algorithm in the discussion for the user study below.

7. 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.

A total of 69 people participated in the study; however, users who didn't answer all the questions, or who provided garbage answers, like "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.

All of these users had been programming in Java for at least 2 years, except 2 people: one person had written a Java program for a class, as well as for a job; the other person had no Java experience at all, but had 20 years of general programming experience.

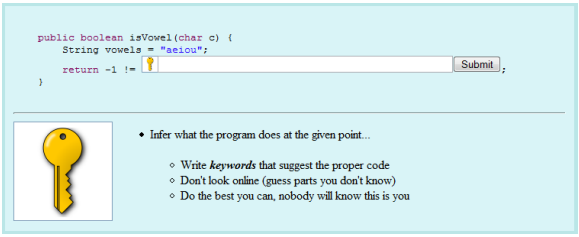

Figure 9: Example of a task used in the user study (task 5 from Table 3). This user is asked to enter keywords that suggest the missing expression, but other users may be asked to enter Java or pseudocode for this task.

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 9. 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 9 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 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

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))

Table 3: Missing Expressions for Tasks

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

Table 4: Response counts and statistics for each instruction type.

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. Task 3 was removed because it requires a literal, which is not handled by our current 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 6).

7.2 Results

Table 4 shows the number of responses for each instruction type, along with various statistics. Note that responses are said to use Java syntax if the response could compile as a Java expression in some context.

When asked to write Java code, users wrote syntactically and semantically correct code 53% of the time. This seems low, but 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.

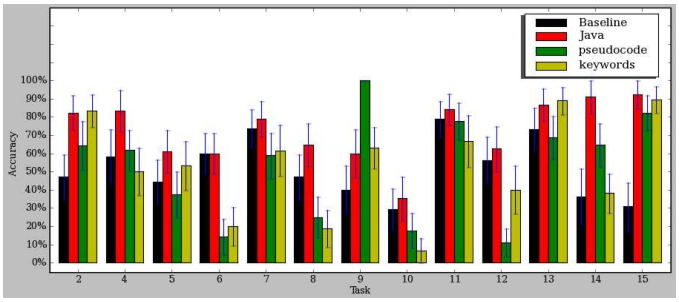

Figure 10: Accuracy of the algorithm for each task, and for each instruction type, along with standard error. The "Baseline" refers to Java responses treated as Java, without running them through the algorithm.

The algorithm translated 59% of the responses to semantically correct Java code. Note that this statistic includes *all* the responses, even the responses 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 responses alone from the baseline 53% to 71%. The accuracies for translating pseudocode and keywords 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 10.

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

7.3 Discussion

It is useful to look at some of the incorrectly translated responses in order to get an idea for where the algorithm fails, and how it could be improved.

7.3.1 A Priori Word Weights

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.

7.3.2 A Priori Function Weights

The response **println f name** in task 10 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

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

Table 5: Responses translated correctly

instructions	translated incorrectly	task
Java	return(x>=0?x;-x);	7
	tokens.append(st.nextToken())	8
	buf.add(s)	11
pseudocode	(x < 0) ? -x : x	7
	lines.append (in.getNext());	12
	input.lowercase();	14
keywords	Add s to buf	11
	in readline insert to lines	12
	print name of f	10

Table 6: Responses translated incorrectly

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.print(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**.

7.3.3 Spell Correction

Many users included **lowercase** in their response to 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`.

7.3.4 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, 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. Overall, the accuracy decreased slightly from 59% to 58%. It may be more effective to create a customized thesaurus for keyword programming, 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.

8. 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 is still for 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 the 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, including method and class signatures [8, 13], specifications [3, 14], metadata attributes [7], identifier usage [10], and documentation comments [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.

9. CONCLUSIONS AND FUTURE WORK

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. These improvements are the primary target of future work. We also plan to test the algorithm on other general purpose languages.

Another important goal of future work is to get field data on the usability of the algorithm. Toward this end, we have already created an Eclipse Plug-in that uses the algorithm to perform keyword query translations in the Java editor, as shown in Figure 1.

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.

10. ACKNOWLEDGMENTS

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.

11. REFERENCES

[1] Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[2] Rosco Hill and Joe 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] Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. *Proceedings of CHI 2007*, to appear.

[5] Greg Little, and Robert C. Miller. Translating Keyword Commands into Executable Code. *Proceedings of User Interface Software & Technology (UIST 2006)*, pp. 135–144.

[6] David Mandelin, Lin Xu, Rastislav Bodik, Doug 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] Naiyana Sahavechaphan and Kajal 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*q, 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.