Type-Directed Completion of Partial Expressions

Daniel Perelman
University of Washington
perelman@cs.washington.edu

Sumit Gulwani Thomas Ball
Microsoft Research Redmond
{sumitg,tball}@microsoft.com

Dan Grossman
University of Washington
djg@cs.washington.edu

Abstract

Modern programming frameworks provide enormous libraries arranged in complex structures, so much so that a large part of modern programming is searching for APIs that "surely exist" somewhere in an unfamiliar part of the framework. We present a novel way of phrasing a search for an unknown API: the programmer simply writes an expression leaving holes for the parts they do not know. We call these expressions *partial expressions*. We present an efficient algorithm that produces likely completions ordered by a ranking scheme based primarily on the similarity of the types of the APIs suggested to the types of the known expressions. This gives a powerful language for both API discovery and code completion with a small impedance mismatch from writing code. In an automated experiment on mature C# projects, we show our algorithm can place the intended expression in the top 10 choices over 80% of the time.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments—Integrated Environments; D.2.13 [*Software Engineering*]: Reusable Software—Reuse Models; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program synthesis

General Terms Languages, Experimentation

Keywords program synthesis, partial expressions, code completion, type-based analysis, ranking

1. Introduction

Modern programming frameworks such as those found in Java and .NET consist of a huge number of classes organized into many namespaces. For example, the .NET Framework 4.0 has over 280,000 methods, 30,000 types, and 697 namespaces. Discovering the right method to achieve a particular task in this huge framework can feel like searching for a needle in a haystack. Programmers often perform searches through unfamiliar APIs using their IDE's code completion, for example Visual Studio's Intellisense, which requires the programmer to either provide a receiver or iterate through the possible receivers by brute force. Fundamentally, today's code completion tools still expect programmers to find the right method by name (something that implicitly assumes they will know the right name for the concept, which may not be true [4]) and to fill in all the arguments.

PLDI'12, June 11–16, 2012, Beijing, China.

Our approach: We define a language of *partial expressions*, in which programmers can indicate in a superset of the language's concrete syntax that certain subexpressions need to be filled in or possibly reordered. We interpret a partial expression as a query that returns a ranked list of well-typed completions, where each completion is a synthesized small code snippet. This model is simple, general, and precisely specified, allowing for a variety of uses and extensions. We developed an efficient algorithm for generating completions of a partial expression. We also developed a ranking scheme primarily based on (sub)typing information to prefer more precise expressions (e.g., a method taking `AVerySpecificType` rather than `Object`).

Code-completion problems addressed: We have used our partial expression language and its implementation for finding completions to address three code-completion problems:

1. Given k arguments and without distinguishing one as the object-oriented receiver, predict a method call including these and possibly other arguments. Note that the name of the method being called is not given; the method name, along with the order of arguments to it, is the output of our system.
2. Given a method call with missing arguments, predict these arguments (with simple expressions such as variables or field and property[1] lookups of variables).
3. Given an incomplete binary expression such as an assignment statement, predict field and property lookups (i.e., given e predict $e.f$) on the left or right side of the operation.

Results: We demonstrate that our approach ranks the correct result highly most of the time and often outperforms or complements existing widely deployed technologies like Intellisense. To collect a large amount of empirical data, we have chosen to leave IDE integration and user studies to future work. Instead, we take existing codebases and run our tool after automatically replacing existing method calls, assignments, and comparisons with appropriate partial expressions. On our corpus of programs, results include:

- For over 80% of method calls (and over 90% if we know the call's return type), there are two or fewer arguments to the call such that with only those arguments, our system will rank the intended method name within the top 10.
- If a simple argument (e.g., a variable) is omitted from a method call, our system can fill it back in correctly (the top-ranked choice) 55% of the time.
- When a field or property lookup is omitted from an expression, we can use surrounding type context to rank the missing property in the top 10 over 90% of the time.

Overall, our work demonstrates that IDEs could use already-available type information to help programmers find methods they want and save keystrokes much more than they do today.

[1] Properties are syntactic sugar for writing getters and setters like fields.

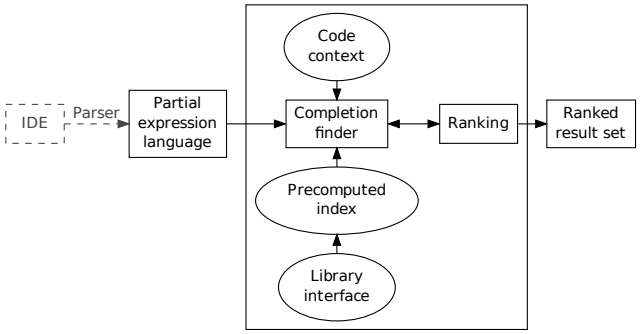

Figure 1. Workflow

Contributions This paper includes the following contributions:

- Identification of the need for a search facility based on partial expressions which we have detailed using illustrative examples in Section 2.
- Design of the partial expressions language and how partial expressions relate to complete expressions as formally defined in Section 3.
- An efficient algorithm for completing partial expressions and ranking the results described in Section 4, which integrates the ranking procedure and off-line indexing of large libraries.
- A large experimental evaluation of the quality and performance of our algorithm on real code presented in Section 5.
- An evaluation of the relative and absolute importance of each ranking feature detailed in Section 5.4.

Section 6 reviews related work and Section 7 concludes the paper.

2. Illustrative Examples

This section describes three examples that use partial expressions in our system and then considers performing the same tasks using prior work. Section 2.1 describes how our system handles these examples. Section 2.2 discusses normal code completion. Section 2.3 covers the closest related work, the Prospector tool[10].

2.1 Our system

Synthesizing Method Names

Suppose you are writing code using an image editing API (specifically, the Paint.NET image editor[2]) and want to figure out how to make an image smaller. Your first instinct may be to write `img.Shrink(size)`. Unfortunately, that API does not exist; the actual API for shrinking an image is

```
public static Document ResizeDocument(
 Document document, Size newSize,
 AnchorEdge edge, ColorBgra background)
```

We will step through how our tool handles this example, using Figure 1 which shows the workflow of our tool. For this example, you would write the query "`?({img, size})`" in the partial expression language described in Section 3. The query is passed to the algorithm represented by the large box described in Section 4 which also has access to the code context which says that `img` and `size` are local variables of types `Document` and `Size`, respectively. The first ten elements of the ranked result set are shown in Figure 2. The static `ResizeDocument` method is the first choice.

Synthesizing Method Arguments

Suppose you already know there is a method `Distance` that returns the distance between two `Point` objects, but are not sure where one of the endpoints is defined. The query "`Distance(point, ?)`" produces a list of `Point`s that could be filled in as the second argument. This includes any locals, fields, or static fields or methods or recursively any fields of those of type `Point`. For example, Figure 3 shows the results of that query in the context of the `EllipseArc` class of the DynamicGeometry library. There, `point` is the only local variable of type `Point`. In this case, the actual argument was `this.Center` which appears third in the list.

[2] `http://www.getpaint.net/`

```
PaintDotNet.Actions.CanvasSizeAction
    .ResizeDocument(img, size, ◇, ◇)
PaintDotNet.Functional.Func.Bind(◇, size, img)
PaintDotNet.Pair.Create(size, img)
PaintDotNet.Quadruple.Create(size, img, ◇, ◇)
PaintDotNet.Triple.Create(size, img, ◇)
PaintDotNet.PropertySystem
    .StaticListChoiceProperty
    .CreateForEnum(img, size, ◇)
System.Drawing.Size.Equals(size, img)
System.Object.ReferenceEquals(size, img)
PaintDotNet.Document.OnDeserialization(img, size)
PaintDotNet.PropertySystem.Property
    .Create(◇, size, img)
```

Figure 2. The first ten results generated and ranked by our system for the query `?({img, size})`.

```
point
this.BeginLocation
this.Center
this.EndLocation
DynamicGeometry.Math.InfinitePoint
shapeStyle.GetSampleGlyph()
.RenderTransformOrigin
this.shape.RenderTransformOrigin
this.ArcShape.Point
this.Figure.StartPoint
this.Shape.RenderTransformOrigin
```

Figure 3. The first ten results generated and ranked by our system for the `?` in the query `Distance(point, ?)`.

Synthesizing Field Lookups

For a more targeted version of the above, the search can be narrowed by specifying the base object to look under. We will consider synthesizing field lookups in the context of a comparison operator. The query "`point.?*m >= this.?*m`" includes `point` and `this` along with zero or more field lookups or zero-argument instance method calls after them. The top ten ranked completions for this query are listed in Figure 4. Note that by completing both holes simultaneously, only completions where the two sides have fields of compatible types are shown.

2.2 Code Completion

Today, programmers can use code completion such as Intellisense in Visual Studio to try to navigate unfamiliar APIs. Intellisense completes code in sections separated by periods ("`.`") by using the type of the expression to the left of the period and textually searching through the list for any string the programmer types. If there is no period, then Intellisense will list the available local variables, types, and namespaces. This often works well, particularly when the programmer has a good idea of where the API they want is or if there are relatively few choices. On the other hand, it performs poorly on our examples.

```
point.X >= this.P1.X
point.X >= this.P2.X
point.X >= this.Midpoint.X
point.X >= this.FirstValidValue().X
point.Y >= this.P1.Y
point.Y >= this.P2.Y
point.Y >= this.Midpoint.Y
point.Y >= this.FirstValidValue().Y
point.X >= this.Length
point.Y >= this.Length
```

Figure 4. The first ten results generated and ranked by our system for the query `point.?*m >= this.?*m`.

Synthesizing Method Names

Using Intellisense to find the nonexistent `Shrink` method, a programmer might type "`img.shr`", see that there is no "`Shrink`" method, and then skim through the rest of the instance methods. As that will also fail to find the desired method, the programmer might continue by typing in "`PaintDotNet.`" and use Intellisense to browse the available static methods, eventually finding `PaintDotNet.Actions.CanvasSizeAction` where the method is located. Hopefully documentation on the various classes and namespaces shown by Intellisense's tooltip will help guide the programmer to the desired method, but this is dependent on the API designer documenting their code well and the documentation using terminology and abstractions that the programmer understands. A programmer would likely search through many namespaces and classes before happening upon the right one.

Synthesizing Method Arguments

If the user has already entered "`Distance(point, `" and then triggers Intellisense, Intellisense will list of every namespace, type, variable, and instance method in context even though many choices will not type-check (as the programmer may intend to call a method or perform a property lookup on one of those objects). When the list is brought up, the most recently used local variable of type `Point`, which would be `point` in this case, will be selected. The programmer will have to read through many unrelated options to locate the other values of type `Point`.

Note that Eclipse's code completion is actually significantly different in this scenario. It will list all of the local variables valid for the argument position along with common constants like `null`. If a more complicated expression is desired, the user has to cancel out and request the normal code completion which is similar to Visual Studio's.

Synthesizing Field Lookups

Given "`point.`", Intellisense will list all fields and methods of that object. The listing will go only one level deep: if the user wants a field of a field, they have to know which field to select first.

2.3 Prospector

The Prospector tool by Mandelin et al.[10] is an API discovery tool which constructs values using mined "jungloids" which convert from one input type to one output type and are combined into longer jungloids. The tool uses a local variable to construct a value of the output type. The motivating example in this prior work is converting an `IFile` to an `ASTNode` in the Eclipse API which requires a non-obvious intermediate step involving a third type:

```
IFile file = ...;
ICompilationUnit cu =
    JavaCore.createCompilationUnitFrom(file);
ASTNode ast = AST.parseCompilationUnit(cu, false);
```

(a) $e ::= call \mid varName \mid e.fieldName \mid e\texttt{:=}e \mid e\texttt{<}e$
$call ::= methodName(e_1, \ldots, e_n)$

(b) $\widetilde{e} ::= \widetilde{a} \mid \texttt{?} \mid \diamond$
$\widetilde{a} ::= e \mid \widetilde{a}\texttt{.?f} \mid \widetilde{a}\texttt{.?*f} \mid \widetilde{a}\texttt{.?m} \mid \widetilde{a}\texttt{.?*m} \mid \widetilde{call}$
$\mid \widetilde{e}\texttt{:=}\widetilde{e} \mid \widetilde{e}\texttt{<}\widetilde{e}$
$\widetilde{call} ::= \texttt{?}(\{\widetilde{e}_1, \ldots, \widetilde{e}_n\}) \mid methodName(\widetilde{e}_1, \ldots, \widetilde{e}_n)$

Figure 5. (a) Expression language (b) Partial expression language

The Prospector UI triggers queries only at assignments to variables, but that is a minor implementation detail.

Synthesizing Method Names

As Prospector can consider only one type as input, a programmer might query for a conversion from `Size` to `Document` or from `Document` to `Document`, which does not quite match the programmers intuition of wanting to resize the document. Prospector will return methods with arguments it cannot fill in. It prefers fewer unknown arguments, so `ResizeDocument` would likely be rather far down in the list of options for either query.

Synthesizing Method Arguments

Queried for type `Point`, Prospector would give a similar list to the one our tool creates, although it does not consider globals as possible inputs to its algorithm. Specifically, Prospector would give any locals of the proper type and recursively find any fields of the proper type. It may also find chains that involve downcasts found to work elsewhere in the codebase, which our tool would not find.

Synthesizing Field Lookups

Prospector does not take suggestions of starting points from the user, although its UI could theoretically be modified to do so. On the other hand, Prospector has only one target type and cannot make more complicated expressions like the one above with a `>=` operator. The closest corresponding use of Prospector would be to guess the type for either side of the comparison and have Prospector find fields of that type.

3. Partial expression language

Queries in our system are partial expressions. A partial expression is similar to a normal (or "complete") expression except some information may be omitted or reordered. A partial expression can have many possible completions formed by filling in the holes and reordering subexpressions in different ways.

Complete expression syntax

Before defining partial expressions, we first define a simple expression language given by the e and $call$ productions in Figure 5(a), which models features found in traditional programming languages. Our simple language has variables, field lookups, assignments, a comparison operator, and method calls. (Other operators are omitted from the formalism.) Also, the receiver of a method call is considered to be its first argument in order to simplify notation as when reordering arguments, an argument other than the first may be chosen as the receiver.

Partial expression syntax

Partial expressions are defined by the $\widetilde{e}$, $\widetilde{a}$, and $\widetilde{call}$ productions in Figure 5(b). Partial expressions support omitting the following classes of unknown information:

- **Entire subexpressions.** `?` gives no information about the structure of the expression, only that it is missing and should be filled in. On the other hand, $\diamond$ should not be filled in: it indicates a

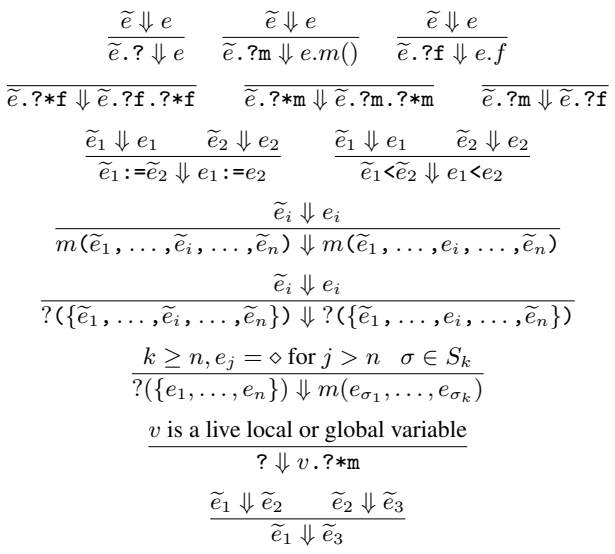

Figure 6. Semantics of partial expressions. The final result must type-check in the context of the query, treating $\diamond$ as having any type.

subexpression to ignore due to being independent of the current query (so making it a `?` would only add irrelevant results) or simply being a subexpression the programmer intends to fill in later, perhaps due to working left-to-right.

- **Field lookups.** The $\widetilde{a}$ production defines a series of four `.?` suffixes which are slightly different ways of saying that an expression is missing one or more field lookups or the desired expression is actually the result of a method call on the expression. The '`f`' suffix is short for "field" and can be completed as a single field lookup or nothing. The `.?*` suffixes complete as the `.?` versions repeated as many times as needed.
- **Simple method calls.** The `.?m` suffix is like the `.?f` suffix. The '`m`' is short for "zero-argument method call" and can be completed as a call to an instance method with zero additional arguments or also as a field lookup or nothing.
- **Which method to call.** `?(`$\{\widetilde{e}_1, \widetilde{e}_2\}$`)` represents a call to some unknown method with two known arguments, which may themselves be partial expressions.
- **Number and ordering of arguments to a method.** For unknown methods, there may also be additional arguments missing or the arguments may be out of order, which is represented by the use of set notation for the arguments in `?(`$\{\widetilde{e}_1, \widetilde{e}_2\}$`)`.

Partial expression semantics

Figure 6 gives the full semantics of the partial expression language. The $\Downarrow$ judgement nondeterministically takes a partial expression to a complete expression with the exception that any $\diamond$ subexpressions remain. With the exception of the `.?*` rules, each rule removes or refines some hole, making the partial expression one step closer to a complete expression. The bottom rule allows for the composition of other rules. The top leftmost rule allows any of the `.?` suffixes to be omitted. For type checking, $\diamond$ is treated as a wildcard: as long as some choice of type for the $\diamond$ works, the expression is considered to type check. The actual algorithm implemented does not use these rules exactly, although it matches their semantics.

The partial expressions language semantics never add operations like multiplication or new method calls (other than to zero-argument methods). The idea is that any place where computation is intended should be explicitly specified, and the completions simply list specific APIs for the computations. The exception for zero-argument methods is made because they are often used in place of properties for style reasons or due to limitations of the underlying language.

Examples

The first example from Section 2, `?({img, size})`, is a method call with an unknown name and two complete expressions as arguments. It can be expanded to any method that can take those two variables in any two of its argument positions, so `Triple.Create(`$\diamond$`, size, img)` is a valid completion. Note that no attempt is made to fill in the extra argument. This is done to reduce the number of choices when recommending methods; for other applications fully completing the expression may be useful. The user may afterward decide to convert the $\diamond$ to `?` or some other partial expression.

Our second example from Section 2, `Distance(point, ?)` can take one step to one of

- `Distance(point, point.?*m)`,
- `Distance(point, this.?*m)`,
- `Distance(point, shapeStyle.?*m)`

or many other possibilities. Any local in scope or global (static field or zero-argument static method) could be chosen to appear before the `.?*m`. Whatever is selected is completed to some expression of type `Point`. Any `.?` suffix can be omitted when completing an expression, so `point.?*m` can be completed as `point` which is the first option in Figure 3. `this.?*m` can also become one or more lookups by going to `this.?m.?*m` in one step and the `.?m` becomes some field. For `ArcShape`, `this.ArcShape.?*m` is further completed to `this.ArcShape.Point`. Any of the completions mentioned so far would have been valid for `.?f` instead of `.?m` as well. On the other hand, for `shapeStyle.?*m`, the first `.?m` from the `.?*m` is completed with an instance method `.GetSampleGlyph()` that returns an object with a field `RenderTransformOrigin` of type `Point` which the remaining `.?*m` can complete to.

An unknown method's arguments may themselves be partial expressions. For example, `?({strBuilder.?*m,e.?*m})` could expand to `Append(strBuilder, e.StackTrace)` (which would normally be written as `strBuilder.Append(e.StackTrace)`).

The third example from Section 2, `point.?*m >= this.?*m`, also uses `.?*m`, so the completions work as above, but, as there are two of them in the expression related by the `>=`, there must be a definition of `>=` which is type compatible with the two completions. In this example, all the comparable fields have types `int` or `double`. But suppose `Point` had a field `Timestamp` of type `DateTime`; then `Point.Timestamp >= this.P1.Timestamp` would be a valid completion, but `Point.X >= this.P1.Timestamp` would not.

4. Algorithm

This section describes an algorithm (represented by the boxed section of Figure 1) for completing partial expressions. The algorithm takes a partial expression and an integer n as input and returns an ordered list of n proposed completions. The algorithm has access to static information about the surrounding code and libraries: the types of the values used in the expression, the locals in scope, and the visible library methods and fields. Bounding n is important because some partial expressions have an infinite list of completions. What constitutes a valid completion is defined by Figure 6.

The algorithm described in this section does completion finding and ranking simultaneously in order to compute the top n completions efficiently. Section 4.1 describes the ranking function. Section 4.2 describes the completion finder and the integrated algorithm whose design is informed by the ranking function.

$$
\begin{aligned}
\mathsf{score}(expr) &= \textstyle\sum_{s \in \mathsf{subexps}(expr)} \mathsf{score}(s) \\
&\quad + \textstyle\sum_{s \in \mathsf{subexps}(expr)} \mathsf{td}(\mathsf{type}(s), \mathsf{type}(\mathsf{param}(s))) \\
&\quad + 2 \cdot \mathsf{dots}(expr) \\
&\quad + \textstyle\sum_{s \in \mathsf{subexps}(expr)} \mathsf{abstype}(s) \\
&\qquad\qquad \neq \mathsf{abstype}(\mathsf{param}(s)) \\
\mathsf{scorec}(call) &= \mathsf{score}(call) \\
&\quad + (\mathsf{isInstance}(call) \vee \mathsf{isNonLocalStatic}(call)) \\
&\quad + \max\Big(0, 3 - (\mathsf{nsArgs}(call) \neq \mathsf{reciever}(call)) \\
&\qquad\qquad \cdot \Big|\textstyle\bigcap_{s \in \mathsf{nsArgs}(call)} \mathsf{ns}(\mathsf{type}(a))\Big|\Big) \\
\mathsf{scorecmp}(expr_1, expr_2) &= \mathsf{score}(expr_1 < expr_2) \\
&\qquad + 3 \cdot (\mathsf{name}(expr_1) \neq \mathsf{name}(expr_2)) \\
\\
\mathsf{nsArgs}(call) &= \{a \in \mathsf{subexps}(call) \mid \mathsf{type}(a) \text{ is not primitive}\}
\end{aligned}
$$

Figure 7. The ranking function. Note that boolean values are considered 1 if true and 0 if false and that abstract types ($\mathsf{abstype}(\cdot)$) are considered not equal if both are $undefined$.

4.1 Ranking

This section defines a function that maps completed expressions that may contain $\diamond$ subexpressions to integer scores. This function is used to rank the results returned by the completion finder in ascending order of the ranking score (i.e., a lower score is better). The function is defined such that each term is non-negative, so if any subset of the terms are known, their sum is a lower bound on the ranking score and can be used to prune the search space.

The computation is a sum of various terms summarized in Figure 7. $\mathsf{score}(\cdot)$ applies to all expressions while $\mathsf{scorec}(\cdot)$ is a specialized version with tweaks for method calls and $\mathsf{scorecmp}(\cdot, \cdot)$ is a specialized version with a tweak for comparisons. The computation is defined recursively, so for methods or operators with arguments, the sum of the scores of their arguments is added to the score. The scoring function incorporates several features we designed based on studying code examples and our own intuition. This section explains these features in detail. Section 5.4 evaluates each feature's contribution to our empirical results.

Type distance The primary feature in the ranking function is "type distance", for example from a method call argument's type to the type of the corresponding method parameter. Informally, it is the distance in the class hierarchy, extended to consider primitive types, interfaces, etc. For example, if `Rectangle` extends `Shape` which extends `Object`, $\mathsf{td}(\texttt{Rectangle}, \texttt{Shape}) = 1$ and $\mathsf{td}(\texttt{Rectangle}, \texttt{Object}) = 2$. Far away types are less likely to be used for each other, so method calls and binary operations where the arguments have a higher type distance are less likely to be what the user wanted.

Formally, the type distance from a type α usable in a position of type β to that type β, $\mathsf{td}(\alpha, \beta)$, is defined as follows:

$$
\mathsf{td}(\alpha, \beta) = \begin{cases} undefined & \text{no implicit conversion of } \alpha \text{ to } \beta \\ 0 & \text{if } \alpha = \beta \\ 1 & \text{if } \alpha \text{ and } \beta \text{ are primitive types} \\ 1 + \mathsf{td}(\mathsf{s}(\alpha), \beta) & \text{otherwise} \end{cases}
$$

$\mathsf{s}(\alpha)$ is the explicitly declared immediate supertype of α which to minimizes $\mathsf{td}(\mathsf{s}(\alpha), \beta)$. Note that $\mathsf{td}(\alpha, \beta)$ is used by the ranking function only when it is defined as that corresponds to the expression being type correct.

The type distance term is the sum of the type distances from the type of each argument arg to the type of the corresponding formal parameter $\mathsf{param}(arg)$. For method calls this is well-defined; binary operators are treated as methods with two parameters both of the more general type, so the type distance between the two arguments to the operator is used.

Depth The next term prefers expressions with fewer subexpressions. The ranking scheme prefers shorter expressions by computing the complexity of the expression which is approximated by the number of dots in the expression and multiplying that value by 2 to weight it more heavily. For example, $\mathsf{dots}(\text{“}\texttt{this.foo}\text{”}) = 1$ so it would get a cost of 2 while $\mathsf{dots}(\text{“}\texttt{this.bar.ToBaz()}\text{”}) = 2$ so it would get a cost of 4. To avoid double counting, any dots which are part of subexpressions are not counted here and instead are included via the subexpressions score term.

In-scope static methods Instance method calls will tend to have a type distance of zero for the receiver, so type distance has an implicit bias against static method calls. Noting that static methods of the enclosing type can be called without qualification, just like instance methods with `this` as the receiver, our ranking algorithm should similarly not disfavor such in-scope static methods. This is fixed by adding a cost of 1 if either the method is an instance method or the method is a static method that is not in scope.

Common namespace As related APIs tend to be grouped into nearby namespaces, the algorithm prefers calls where the types of all the arguments with non-primitive types and the class containing the method definition are all in the same namespace. Primitive types, including `string`, are ignored in this step because they are used with varying semantics in many different libraries. Furthermore, deeper namespaces tend to be more precise so a deep common namespace indicates the method is more likely to be related to all of the provided arguments. Specifically, the algorithm takes the set of all namespaces of non-primitive types among the arguments, treats them as lists of strings (so "`System.Collections`" is ["`System`", "`Collections`"]), finds the (often empty) common prefix, and uses its length to compute the "namespace score". To avoid this boosting the scores of instance calls with only one non-primitive argument, the similarity score is 0 in that case.

In order to have the namespace similarity term be non-negative, namespace similarities are capped at 3, and 3 minus the length of the common prefix is used as the common namespace term.

Same name Comparisons are often made between corresponding fields of different objects. Whether two fields have corresponding meanings can be approximated by checking if they have the same name. That is, `p.X` is more likely to be compared to `this.Center.X` than to `this.Center.Y`. To capture this, a 3 point penalty is assigned to comparisons where the last lookups on the two sides do not have the same name. The value is intentionally chosen to be greater than the cost of a lookup, so a slightly longer expression that ends with a field of the right name is considered better than a shorter one that does not.

Abstract type inference

We now introduce an important refinement to the basic ranking function that partitions types into "abstract types" based on usage, which is particularly important for commonly used types like `string`. Abstract types may have richer semantics than `string` such as "path" or "font family name". Our approach is based on the Lackwit tool that infers abstract types of integers in C[11].

Abstract types are computed automatically using type inference. An abstract type variable is assigned to every local variable, formal parameter, and formal return type, and a type equality constraint is added whenever a value is assigned or used as a method call argument. As all constraints are equality on atoms, the standard unification algorithm can be implemented using union-find.

In order to avoid merging every abstract type `.ToString()` or `.GetHashCode()` is called on, methods defined on `Object` are

treated as being distinct methods for every type. All other methods have formal parameter and formal return type terms associated with their definition which are shared with any overriding methods. A more principled approach might involve a concept of subtyping for abstract types, but that would greatly complicate the algorithm for what would likely be minimal gain.

For example, consider the following code from Family.Show:[3]

```
string appLocation = Path.Combine(
    Environment.GetFolderPath(
        Environment.SpecialFolder.MyDocuments),
    App.ApplicationFolderName);
if (!Directory.Exists(appLocation))
    Directory.CreateDirectory(appLocation);
return Path.Combine(appLocation,
                    Const.DataFileName);
```

`Directory.Exists`, `Directory.CreateDirectory`, and `Path.Combine` take `appLocation` as their first argument, so the analysis concludes their first arguments are all the same abstract type. Furthermore, from the first statement, that must also be the abstract type of the return values of `Path.Combine` and `Environment.GetFolderPath`. On the other hand, there is no evidence that would lead the analysis to believe the second argument of `Path.Combine` is of that abstract type, instead `App.ApplicationFolderName` and `Const.DataFileName` are believed to both be of some other type. Intuitively, a programmer might call those two types "directory name" and "file name".

In the ranking function, the type distance computation is refined by adding an additional cost of 1 if the abstract types do not match.

4.2 Completion finder

This section presents a general algorithm for computing the top n ranked completions of a partial expression, first giving a naive implementation and then discussing optimizations. The main logic is Algorithm 1 which returns a generator that returns all completions of a partial expression in order by score. The **yield return** statement returns a single completion, and when the next completion is requested, execution continues on the next statement. The first n elements of `AllCompletions(`$\widetilde{e}$`)` are the top n ranked completions of $\widetilde{e}$.

Note that for any partial expressions containing `.?*f` or `.?*m`, this generator will usually continue producing more completions forever, but can be called only n times to get just the top n completions. It may be easier to first read the algorithm while ignoring the `score` variable, which is necessary to handle the unbounded result set for `.?*f` or `.?*m`. Then it is a simple recursive algorithm which computes every possible completion of its subexpressions and uses those completions to generate every possible completion of the entire expression (e.g. all methods that can take those arguments).

Note that `?` is interpreted as $vars$`.?*m` where $vars$ is a special subexpression whose list of completions is every local and global variable in scope.

We now discuss various useful optimizations.

Cache subexpression scores

A subexpression's score will be needed for every completion it appears in. To compute it only once, the algorithm is redefined such that it returns a set of pairs of completions and their scores.

Compute completions not in score order

In the algorithm given above, completions with a score not equal to `score` are discarded and regenerated later. To avoid that work, completions with a score greater than `score` can be saved to be output later. For most partial expressions, it makes sense to compute all of the completions for a given value of `concreteSubs` at once. In the case of `.?*f` and `.?*m` queries, the algorithm will never be done computing every possible completion, but `foo.?*f` can be thought of as the union of `.?f` queries first on `foo`, then on the results of `foo.?f`, etc. Then each completion set is finite and the basis for the future completions. In pseudocode, this is implemented as inserting each completion `c` into `subcomps`.

[3] `http://www.vertigo.com/familyshow.aspx`

Algorithm 1: `AllCompletions(`$\widetilde{e}$`)`

```
input  : ẽ: a partial expression
output : a generator of all completions in order by score
subexps ← the list of immediate subexpressions of ẽ;
Let subcomps be a map from subexps to completions;
foreach s ← subexps do
    subcomps[s] ← AllCompletions(s);
end
foreach score ← [0, ∞) do
    foreach concreteSubs ← all choices of exactly one
    completion for each subexpression from subcomps
    whose score(concreteSubs) ≤ score do
        foreach type-correct completion c of ẽ using
        subexpressions concreteSubs where
        score(c) = score do
            yield return c;
        end
    end
end
```

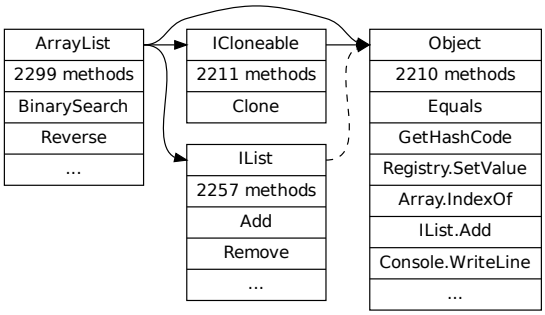

Figure 8. The method index. The supertypes of `IList` other than `Object` are omitted for brevity.

Indexing

As written, how the algorithm iterates over possible completions is unspecified. This is especially a problem for unknown methods as simply iterating over all methods in a huge framework would take too long. An index is maintained that maps every type to a set of methods for which at least one of the arguments may be of that type. Then, given a query like `?(`$\{e_1, e_2\}$`)`, each of the argument types is looked up to see how many methods would have to be considered for that type and the smallest set is chosen. That set will almost always be orders of magnitude smaller than the set of all methods.

Part of a method index is shown in Figure 8. In order to save memory, the method index is organized such that looking up a type τ gets a set of methods which have parameters of the exact type τ along with pointers to the method indexes for the immediate supertypes of τ. Due to the type distance part of the ranking algorithm explained in Section 4.1, each method index visited will give progressively worse ranked results.

Methods are a prime candidate for indexing as there are many methods and few that take a specific type. Although the current implementation does not do so, queries for multiple field lookups could also be made more efficient using an index that indicates for each type which types are reachable by a `.?*f` or `.?*m` query, how many lookups are needed, and which lookups can lead to a value of that type. For example, a `Line` type with `Point` fields `p1` and `p2` and a `GetLength()` method would have an entry denoting that the type `double` is reachable in ≥ 2 lookups using a `.?*f` query with the next lookup being one of `p1` or `p2` while it is reachable in ≥ 1 lookup using a `.?*m` query with the next lookup being one of `GetLength()`, `p1`, or `p2`.

Avoid computing type-incorrect completions

If the possible valid types for the completions were known, then the type reachability index would be more useful: otherwise results of every type have to be generated anyway for completeness. At the top level, the context will often provide a type unless the expression being completed is the initial value for a variable annotated only as `var`. On the recursive step of the algorithm, the possible types may not be known or may not be precise enough to be useful: there are methods that take multiple arguments of type `Object`, so even knowing one of the argument types does not narrow down the possibilities for the rest. On the other hand, binary operators and assignments are relatively restrictive on which pairs of types are valid, so enumerating the types of the completions for one side could significantly narrow down the possibilities for the other side.

Grouping computations by type

Which completions are valid is determined solely by the types of the expressions involved. Hence, instead of considering every completion of every subexpression separately, the completions of each subexpression can be grouped by type after grouping by score to reduce the number of times the algorithm has to check if a given type is valid in a given position. This also allows type distance computations to be done once for all subexpressions of the same types. Any remaining ranking features are computed separately for each completion as grouping by them is no faster than computing their terms of the ranking function.

5. Evaluation

We implemented the algorithm described using the Microsoft Research Common Compiler Infrastructure (CCI).[4] CCI reads .NET binaries and decompiles them into a language resembling C#. Unfortunately, we were unable to work on actual source code because at the time the experiments were performed, no tools for analyzing the source code of C# programs existed—and even if they did exist, open source C# programs are relatively rare.

We performed experiments where our tool found expressions in mature software projects, removed some information to make those expressions into partial expressions, and ran our algorithm on those partial expressions to see where the real expression ranks in the results.

All experiments were run on a virtual machine allocated one core of a Core 2 Duo E8400 3GHz processor and 1GB of RAM.

One minor issue is that any precomputation, specifically abstract type inference, would see the expression we are trying to recreate when in actual practice the expression would not yet exist. To avoid this situation, we re-run abstract type inference for each expression, eliminating the expression and all code that follows it in the enclosing method—we do consider the rest of the program.

We describe three case studies whose significance we have previously discussed in Section 2 and show that the ranking scheme is effective and the algorithm is efficient. After that, we analyze the importance of the individual ranking features in Section 5.4. Finally, we discuss threats to validity in Section 5.5.

[4] `https://cciast.codeplex.com/`

Table 1. Summary of quality of best results for each call

Program	# calls	# top 10	# top 10..20
Paint.Net[a]	3188	2288	525
WiX[b]	13192	11430	512
GNOME Do[c]	208	167	22
Banshee[d]	91	82	2
.NET[e]	2801	2345	145
Family.Show[f]	586	510	23
LiveGeometry[g]	1110	1072	3
Totals	21176	17894 (84.5%)	1232 (5.8%)

[a] `http://www.getpaint.net/`—image editor (main `.exe`)
[b] `http://wix.codeplex.com/`—Windows Installer XML
[c] `http://do.davebsd.com/`—application launcher
[d] `http://banshee.fm/`—media player
[e] .NET Framework v3.5 libraries System.Core.dll, mscorlib.dll
[f] `http://www.vertigo.com/familyshow.aspx`—WPF example application
[g] `http://livegeometry.codeplex.com/`—geometry visualizer

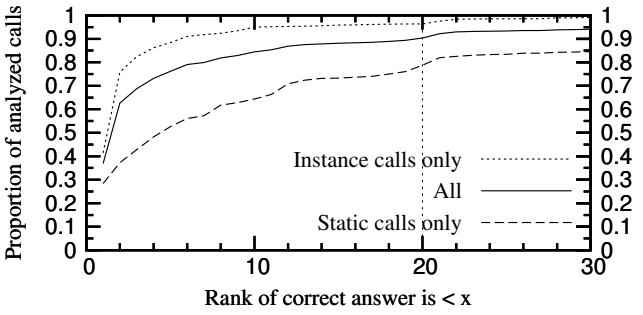

Figure 9. The proportion of calls of each type with the best rank of at least the given value

5.1 Predicting Method Names

Our first experiment shows that queries consisting of one or two arguments can effectively find methods. We ran our analysis on 21,176 calls across parts of seven C# projects listed in Table 1. We generated queries by finding all calls with ≥ 2 arguments (including the receiver, if any) and giving one or two of the call's arguments to the algorithm. We evaluated the algorithm on where in the results list the actual method appeared. While putting the correct result as the first choice is ideal, we do not consider it necessary for usefulness since users can quickly skim several plausible results.

Figure 9 shows the results overall and partitioned between static and instance calls. Almost 85% of the time, the algorithm is able to give the correct method in the top 10 choices. An additional 5% of the time, the correct method appears in the next 10 choices out of a total of hundreds of choices on average.

Notably, the algorithm fares significantly better on instance calls than static calls. This is not too surprising as the search space is much larger for static calls. This might also indicate that the current heuristics prefer instance calls more strongly than they should.

Unfortunately, we cannot algorithmically determine which argument subset a user would use as their search query. Instead, we show that usually for *some* set of no more than 2 arguments the correct method being highly ranked. Our intuition, which would need a user study to validate fully, is that evaluating our approach

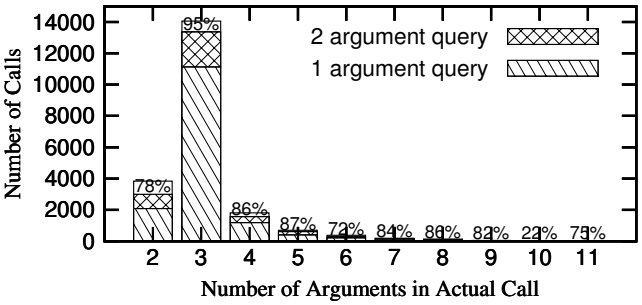

Figure 10. Each bar represents all calls analyzed with the number of arguments. The sections of the bar correspond to how many of those arguments the algorithm needed to put the original call in the top 20 results.

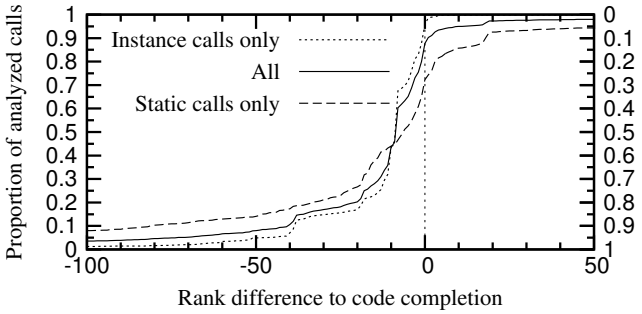

Figure 11. Difference in rank between our algorithm and Intellisense

by choosing for the best possible subset of arguments is reasonable because programmers are capable of identifying the most useful arguments (e.g., `PreciseLibraryType` instead of `string` or `Pair`).

Figure 10 shows that a single argument is often enough for the algorithm to determine which method was desired, in this case defined as putting the method in the top 20 choices. Not shown in the graph is that adding a third argument leads to only negligible improvement in these results; note that even knowing all of the arguments to a method might not be enough to place it in the top 20 choices. Above the bars is the percentage of calls the algorithm was able to guess using only two arguments, which is high for any number of arguments. The intuition is that most of the arguments are not important, although there are also more opportunities for an argument to be of a rarely used type. The low value for 10 argument calls is due to there being very few such calls and most of them being from a large family of methods which all have the same method signature.

Comparison to code completion

Figure 11 compares our ranking algorithm to Intellisense. The y-axes are read as the left side measuring the proportion of calls our system did better on and the right side measuring the proportion of calls Intellisense did better on.

We modeled Intellisense as being given the receiver (or receiver type for static calls) and listing its members in alphabetic order. Intellisense knows which argument is the receiver but is not using knowledge of the arguments. It was considered to list only instance members for instance receivers and only static members for static receivers. Given this ordering, we were able to compute the rank in the alphabetic list of the correct answer. We then subtracted that rank from the rank given by our algorithm, so negative numbers mean our algorithm gave the correct answer a higher rank.

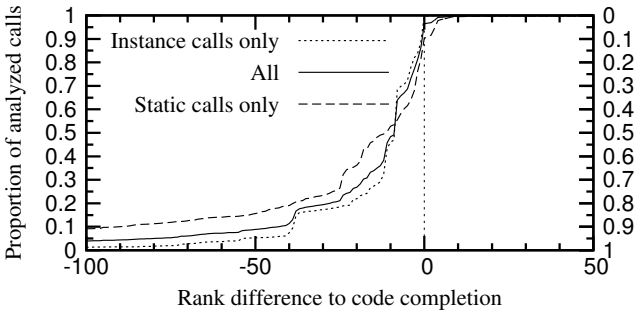

Figure 12. Difference in rank between our algorithm filtering its results for those matching the correct return type and Intellisense

About 45% of the time, our position is at least 10 higher than it is with Intellisense. Since Intellisense displays at most 10 results at a time, this means it is not initially displayed by Intellisense.

Subtracting the ranks is oversimplifying the comparison. First, the different tools have different inputs. Our tool does not require the receiver but is helped by being provided a second argument. Second, the Intellisense results are listed in alphabetic order which is likely easier to skim through than the results from our tool which will be ordered by their ranking scores. The take-away is not that our tool is "better" than Intellisense; they serve different purposes. Instead, we wanted to show that our tool is often able to greatly reduce the number of choices a user would have to sift through compared to Intellisense even if the user knew the correct receiver.

Figure 12 shows a similar comparison to the one in Figure 11 except that our algorithm additionally knew the desired return type (or `void`) and only suggested methods whose return type matched.

The assumption of a known return type is not used elsewhere both because, in the context of API discovery, the user may often not know what return type to use, and the `var` keyword in C# and equivalents in other languages allow a user to omit return types.

Speed For 98.9% of the calls analyzed, the query with the best result ran in under half a second, which is fast enough for interactive use.

As a caveat, these times do not include running the abstract type inference algorithm. That could take as long as several minutes for a large codebase but can be done incrementally in the background.

These times were measured using CCI reading binaries as opposed to getting the information from an IDE's incremental compiler. How that affects performance is unclear, but any such effects were minimized by memoizing a lot of the information from CCI, so the vast majority of the time was spent in our algorithm.

5.2 Predicting Method Arguments

Our second experiment investigated how often arguments to a method could be filled in by knowing their type.

Looking at the same method calls as the previous experiment, for each argument in each call, a query was generated with that argument replaced with `?`. There were a total of 69,927 arguments across the 21,176 calls. 23,927 were considered not guessable due to having an expression form that our partial expression completer does not generate like an array lookup or a constant value.

Figure 13 shows how well our algorithm is able to predict method arguments, with the lower line ignoring the low-hanging fruit of local variables. Over 80% of the time, the algorithm is able to suggest the intended argument as one of the top 10 choices given out of an average of hundreds of choices.

Use of expressions other than local variables in argument positions is common as shown in Figure 14. Programmers must somehow discover the proper APIs for these expressions: Intellisense

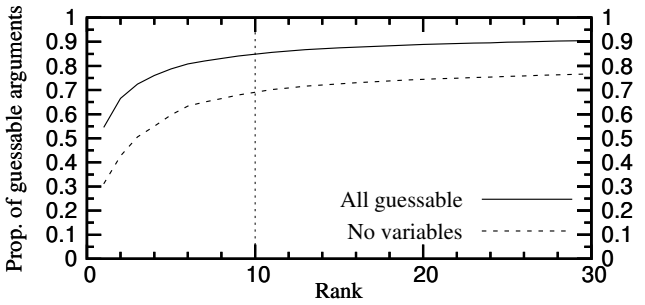

Figure 13. The proportion of guessable arguments which could be guessed with a given rank and the same ignoring arguments which are just local variable references

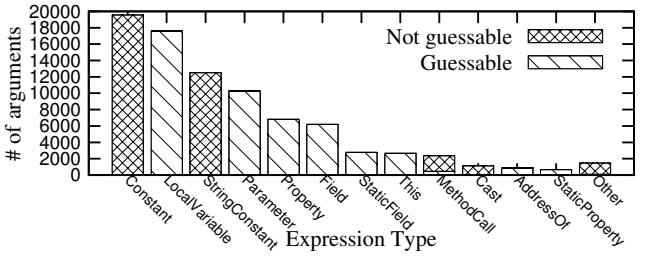

Figure 14. The types of expressions found in argument positions, showing which ones are considered guessable

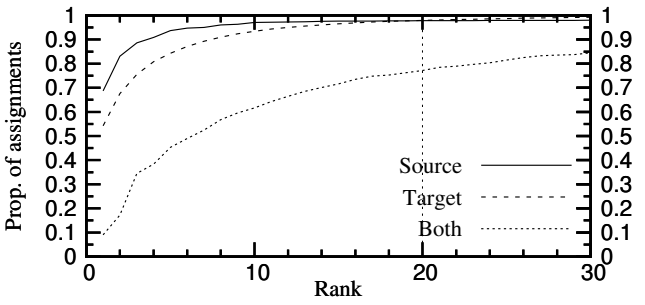

Figure 15. Proportion of assignments where a field lookup could be removed from one or both sides and guessed with a given rank

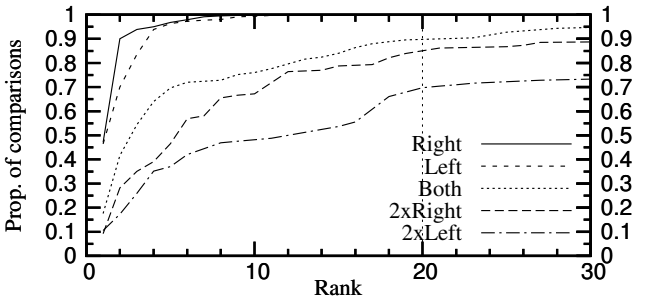

Figure 16. Proportion of comparisons where field lookups could be removed from one or both sides and guessed with a given rank

only suggests local variables given an argument position. The "not guessable" expressions are those that involve constants or computation like an addition or a non-zero argument method call that could be guessed by neither our technique nor Intellisense. Our partial expression language captures more of the expressions programmers use as arguments including field/property lookups which are relatively common and require browsing to find using Intellisense.

As our experiments are on decompiled binaries and not the original source, these arguments may not be exactly what the programmer wrote. In particular, expressions might be stored in temporary variables that they did not write or temporaries they did write might be removed, putting their definition in an argument position.

Speed Our tool is capable of enumerating suggested arguments in under a tenth of a second 92% of the time and under half a second over 98% of the time, which is fast enough for an interactive tool.

5.3 Predicting Field Lookups

Our third experiment determines how often field/property lookups could be omitted in assignments and comparisons (on either side).

Our corpus includes 14,004 assignments where the target ends with a field lookup, 7,074 where the source does, and 966 where both do. For those assignments, Figure 15 shows the rank of the correct answer when our algorithm was given the assignment with the final field lookups removed and `.?m` added to the end of both sides of the assignment. The correct answer was in the top 10 choices over 90% of the time when one field lookup was removed, but only about 59% of the time when a field lookup was removed from both sides, going up to 75% when considering the top 20 choices. There were a total of dozens of choices on average.

Our corpus includes 620 comparisons where the left side ends in a lookup, 162 of which end in two lookups; 620 comparisons where the right side ends in a lookup, 174 of which end in two lookups; and 125 where both sides end in a lookup. Of those, Figure 16 shows the ranks our tool gave to the expression in the source given the query containing the original expression with the lookups removed and `.?m.?m` added to the end of both sides.

The numbers are significantly better than for assignments because there are fewer possibilities: few types support comparisons. One lookup can be placed within the top 10 for nearly every instance in our corpus. If we allow 20 choices, then two lookups where one lookup is on each side can be guessed 89% of the time while two lookups on the same side can be guessed 85% of the time if they are are on right and 69% of the time of they are on the left. The discrepancy between the latter two appears to be that comparisons against constants are usually written with the complicated expression on the left and the constant on the right, and the name matching feature is not helpful for comparisons to constants.

Speed 99.5% of these queries ran in under half a second.

5.4 Sensitivity analysis of ranking function

To see which parts of the ranking function were most important, we re-ran the experiments with various modified ranking functions. Each modified version either included only one of the terms or left out one of the terms, in addition to versions that left out and included both the type distance term and the abstract type term. Table 2 shows the data for the proportion of expressions where the correct answer was in the top 20 choices for different variants of the ranking function for each of the experiments.

Methods Type distance and abstract type distance are the only features that matter. Leaving out the namespace and in-scope static terms seems to make almost no difference. Furthermore, the two type distance terms separately are both good, with abstract type distance alone being a little better, but not quite as good as the two together, confirming that both are useful.

Arguments It seems that only the depth feature seems to matter. Leaving it out makes the results much worse while leaving any other term out has almost no effect. In fact, looking at the "+d"

	Count	All	−n	−s	−d	−m	−t	−a	−at	+n	+s	+d	+m	+t	+a	+at
Methods																
All	21176	0.90	0.90	0.90	-	-	0.85	0.84	0.43	0.43	0.43	-	-	0.84	0.85	0.90
Instance	13904	0.96	0.96	0.96	-	-	0.87	0.94	0.31	0.31	0.31	-	-	0.94	0.87	0.96
Static	7272	0.78	0.78	0.78	-	-	0.80	0.65	0.65	0.68	0.65	-	-	0.64	0.81	0.78
Arguments																
Normal	45325	0.90	0.90	-	0.72	-	0.90	0.90	0.90	0.72	-	0.90	-	0.72	0.72	0.72
No variables	14925	0.77	0.77	-	0.65	-	0.76	0.77	0.76	0.64	-	0.76	-	0.65	0.64	0.65
Assignments																
Target	14004	0.97	-	-	0.87	-	0.97	0.97	0.97	-	-	0.97	-	0.81	0.87	0.87
Source	7074	0.89	-	-	0.87	-	0.90	0.89	0.90	-	-	0.90	-	0.87	0.89	0.87
Both	966	0.75	-	-	0.76	-	0.74	0.75	0.73	-	-	0.73	-	0.75	0.75	0.76
Comparisons																
Left	620	1.00	-	-	0.23	1.00	1.00	1.00	1.00	-	-	1.00	0.65	0.25	0.68	0.25
Right	620	1.00	-	-	0.51	1.00	1.00	1.00	1.00	-	-	1.00	0.53	0.62	0.85	0.62
Both	125	0.89	-	-	0.46	0.87	0.88	0.89	0.88	-	-	0.87	0.46	0.42	0.37	0.42
2xLeft	162	0.69	-	-	0.14	0.69	0.70	0.69	0.70	-	-	0.69	0.41	0.18	0.57	0.18
2xRight	174	0.85	-	-	0.63	0.81	0.81	0.85	0.81	-	-	0.77	0.58	0.71	0.73	0.71

Table 2. Ranking function term sensitivity. Each cell is the proportion where correct answer was found in the top 20 choices with various modifications of the ranking function. "All" is the full ranking function. For the rest, − means without certain terms, + means with only certain terms: 'n'=namespaces, 's'=in-scope static, 'd'=depth, 'm'=matching name, 't'=normal type distance, and 'a'=abstract type distance.

column, using just the depth term gives almost exactly the same results as the full ranking scheme.

Assignments For just one lookup removed from either side, once again only depth matters, but when a lookup is missing from both sides, the type distance computation becomes important. In fact, in the case of a lookup missing from both sides, leaving out the depth component improves the results. This is not too surprising as there are likely many possible assignments which require adding only one lookup to either side. The interesting part is that apparently these lookups can be distinguished from the proper one by looking at more detailed type information (recall that only assignments which are type correct are even being considered).

Comparisons Depth once again seems to be most important. Except on the "2xRight" row, depth appears to be the only significant feature. The different values for that row vary little, indicating that each ranking feature is somewhat useful, but there is little gain from combining them. On average, there were hundreds of type-correct options, so the ranking function is definitely doing something to place the correct option in the top 20.

5.5 Threats to validity

As the experiments were run on decompiled code instead of actual source code, they may not apply to how programmers actually write code. Particularly, the decompiled code may have simpler expressions due to a compiler factoring out additional local variables from complex expressions. This did not appear to be the case from looking at code being analyzed as non-trivial expressions were visible like method calls and binary operations in method argument positions. On the other hand, the decompiled code looked different across projects (specifically, some projects had local variables with names like "`local0`" while others had actual names), so there may be some compiler-dependence involved.

Working on completed projects as opposed to codebases in the process of being developed means that abstract type inference algorithm may have had more information than it would have had in a real development scenario. Note that since the rest of the features work only on the current expression, they are unaffected by the maturity of the codebase. On the other hand, more development likely corresponds to more APIs existing in the codebase which could appear in the results of a query.

The subset of the arguments that got the best results was always used, not the subset of arguments that the programmer would be most likely to think of when writing a query. Naturally, the latter is difficult to determine automatically. Looking at the results, the queries generated tended to seem reasonable. The main reason this could be a problem is when one of the options to a method is a type used only as an option to a small number of methods like the `System.IO.FileMode` enum which is used only for methods that open files. Such types seemed to be rare, although no quantitative analysis was done to confirm that.

More importantly, every single expression in the analyzed codebases was used, as opposed to restricting the analysis to only the ones that the programmer would find difficult to write. Naturally, the latter is difficult to determine automatically. It seems safe to assume that the vast majority of API usages in a codebase were easy for the programmer to write, so the experiment results cannot be taken as direct evidence that our proposed tool would be effective in answering queries. Instead, the experiment results should be taken as an argument that expressions in a program tend to be well specified by a small subset of the information needed to exactly define the expression, and therefore the technique of searching for expressions using a partial expression as a query should tend to work well in practice.

The experiments worked only on C# code. We expect to obtain similar results on code written in languages with a rich type system such as Java.

6. Related Work

Prospector[10] is perhaps the closest related work. With Prospector, as discussed in Section 2.3, a user makes a query for a conversion from one type to another and gets what the authors call a "jungloid" which is a series of operations including method calls, field lookups, and downcasts from examples in the code. That paper noted that shorter jungloids tend to be more likely to be correct and also that jungloids that cross package boundaries are less likely to be correct, both of which are ideas used by our ranking function.

PARSEWeb[17] performs the same task as Prospector except it mines code examples from web searches.

InSynth[6] also produces expressions for a given point in code using the type as well as the context to build more complicated expressions using a theorem prover. InSynth's ranking algorithm

is based on machine learning from examples. It, like Prospector, differs from our work in that it generates expressions from scratch with no input from the programmer to guide it. Their evaluation was on small snippets of Java example code translated to Scala which is difficult to compare to our evaluation on mature C# projects.

Typsy[1] searches for APIs by generating expressions involving any number of method and constructor calls and field lookups given a list of arguments, a return type, and a library package to search within. Typsy will only return expressions with all arguments filled in, so it will generate expressions to construct any missing arguments for methods it finds. The expressions are ranked by their size similar to the depth term in our system, although our depth term does not count the number of method and constructor calls because it will not generate new ones not specified by the programmer.

API Explorer[3] supports queries both for methods taking a given argument and for how to construct a value of a given type. When querying for methods, a keyword can also be provided, so the results are filtered to contain only methods with synonyms of that keyword in the name. The ordering of the results uses a computation similar to our type distance feature. Both types of queries can be expressed in our partial expressions language, except we do not have a way to filter by keywords, and the version used for our experiments does not generate constructor calls when asked for an unknown method.

Strathcona[7] and XSnippet[14] both use context to produce queries which may be helpful to the programmer. In a similar vein, CodeBroker[18] performs searches for APIs based on the documentation of the method currently being written in order to recommend APIs the user may not even be aware of.

Little and Miller[9] propose a system using "keyword programming" to generate method calls where the user gives keywords and the system generates a method call that includes arguments that have most or all of the keywords. Their system attempts to be closer to natural language than ours at the cost of a lower success rate.

Hou and Pletcher[8] recommend various ways of filtering and sorting APIs to show the most relevant choices first. Their system is mainly based on manual annotations of APIs as well as considering how often an API is used. Their work is complementary as it could be used to filter out irrelevant methods from our results.

Searching for functions by type has been recommended for functional programming languages[13][20]. Those proposals differ in that the type signature alone, along with modifications to, for example, handle both curried and uncurried functions, tends to be sufficient for a search. In imperative languages with subtyping, inexact matches are more likely to be meaningful and side-effects make it more likely that many options have the same type.

For discovery of entire modules at once, specification matching can search by specification[21]. Semantics-based code search[12] similarly searches based on specifications including tests and keywords but additionally may make minor modifications to the code to fit the details of the specification.

SNIFF[2] returns snippets matching natural language queries by mining multiple examples from existing code, matching them based on the documentation of the APIs they use, and combining them based on their similarities to eliminate the usage-specific parts of the snippets. Unlike our algorithm, this technique requires the API being searched to be well-documented.

MatchMaker[19] handles API discovery at a different scale: given two types, it generates the glue code to connect those two types by generalizing examples from existing code.

Program sketching[15][16] is a form of program synthesis[5] where the programmer writes a partial program with holes and provides a specification the solution must satisfy. Our technique is similar but considers only a single expression at a time and avoids the need for an explicit specification by using type information to filter the results.

7. Conclusions and Future Work

This paper has shown that type-directed completion of partial expressions can effectively fill in short code snippets that are complicated enough to be difficult to discover using code completion. Furthermore, our ranking scheme is able to sift through hundreds of options to often place the correct answer among the top results.

Future work would be to implement an IDE plug-in and perform a user study to determine if it is useful in real development situations as well as possibly seeing if developers have other ideas for how such a plugin could be used or for similar ideas for lightweight searches. Extending the algorithm to other programming languages is also future work. The features are at least partially tied to C#/Java and will need to be adapted to make sense in other languages.

Acknowledgments

This work was funded by a grant from Microsoft Research. We thank the anonymous reviewers for their valuable feedback.

References

[1] C. Bolton. Typsy: a type-based search tool for Java programmers. In C. Miller, editor, *Selected 2001 SRC Summer Intern Projects*, Technical Note 2001-004. Compaq Systems Research Center, Dec. 2001.

[2] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A search engine for Java using free-form queries. ETAPS/FASE, 2009.

[3] E. Duala-Ekoko and M. P. Robillard. Using structure-based recommendations to facilitate discoverability in APIs. ECOOP, 2011.

[4] G. Furnas, T. Landauer, L. Gomez, and S. Dumais. The vocabulary problem in human-system communication. *CACM*, 30, Nov 1987.

[5] S. Gulwani. Dimensions in program synthesis. In *PPDP*, 2010.

[6] T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. In *Computer Aided Verification (CAV) Tool Demo*, 2011.

[7] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. ICSE, 2005.

[8] D. Hou and D. M. Pletcher. Towards a better code completion system by API grouping, filtering, and popularity-based ranking. RSSE, 2010.

[9] G. Little and R. C. Miller. Keyword programming in Java. ASE, 2007.

[10] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. PLDI, 2005.

[11] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. ICSE, 1997.

[12] S. P. Reiss. Semantics-based code search. ICSE, 2009.

[13] M. Rittri and M. Rittri. Retrieving library identifiers via equational matching of types. CADE, 1992.

[14] N. Sahavechaphan and K. Claypool. XSnippet: mining for sample code. OOPSLA, 2006.

[15] A. Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[16] A. Solar-Lezama. The sketching approach to program synthesis. APLAS, 2009.

[17] S. Thummalapenta and T. Xie. PARSEWeb: a programmer assistant for reusing open source code on the web. ASE, 2007.

[18] Y. Ye and G. Fischer. Supporting reuse by delivering task-relevant and personalized information. ICSE, 2002.

[19] K. Yessenov, Z. Xu, and A. Solar-Lezama. Data-driven synthesis for object-oriented frameworks. OOPSLA, 2011.

[20] A. M. Zaremski and J. M. Wing. Signature matching: a tool for using software libraries. *ACM Trans. Softw. Eng. Methodol.*, April 1995.

[21] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Trans. Softw. Eng. Methodol.*, 6:333–369, 1996.