

How Do Programmers Express High-Level Concepts using Primitive Data Types?

1st Yusuke Shinyama
dept. name of organization (of Aff.)
Tokyo Institute of Technology
Meguro-ku, Tokyo, Japan
euske@sde.cs.titech.ac.jp

2nd Yoshitaka Arahori
dept. name of organization (of Aff.)
Tokyo Institute of Technology
Meguro-ku, Tokyo, Japan
arahori@c.titech.ac.jp

3rd Katsuhiko Gondow
dept. name of organization (of Aff.)
Tokyo Institute of Technology
Meguro-ku, Tokyo, Japan
gondow@cs.titech.ac.jp

Abstract—We investigated how programmers express high-level concepts such as path names and coordinates using primitive data types. While relying too much on primitive data types is sometimes criticized as a bad smell, it is still a common practice among programmers. We propose a novel way to accurately identify expressions for certain predefined concepts by examining API calls. We defined twelve conceptual types used in the Java Standard API. We then obtained expressions for each conceptual type from 26 open source projects. Based on the expressions obtained, we trained a decision tree-based classifier. It achieved 83% F-score for correctly predicting the conceptual type for a given expression. Our result indicates that it is possible to infer a conceptual type from a source code reasonably well once enough examples are given. The obtained classifier can be used for potential bug detection, test case generation and documentation.

Index Terms—Program comprehension, Software maintenance, Source code analysis, Dataflow analysis, Conceptual types

I. INTRODUCTION

Today, the benefits of type system in programming languages are well understood. Since a well-defined type system can prevent a programmer from doing certain invalid operations, it helps a programmer to achieve the correctness and safety. In a statically typed language, proper typing also helps maintenance as it indicates the programmer’s intention. However, defining a domain-specific type system for every concept in a program is cumbersome. At some point, a programmer has to rely on a more primitive data type that is closer to the runtime environment.

a) Program with “primitive obsession”:

```
String username = getCurrentUserName();
String path = "/home/" + username + "/user.cfg";
// Unsafe path: extra check is needed!
File config = new File(path);
```

b) Equivalent well-typed program:

```
User user = getCurrentUser();
Path path = Paths.get(
    user.getHomeDirectory(), "user.cfg");
// Path is guaranteed to be safe.
File config = new File(path);
```

Fig. 1. Primitive obsession (Java)

In fact, programmers tend to use a lot of primitive data types for a variety of purposes. In particular, string and

integer are among most commonly used data types in modern programming languages. A string variable, for example, can be used for storing any text content, such as user name, address and phone number. Strings are so versatile that some programming languages only support the string data type [1]. An integer is also versatile in that it can be used for a size, counter, index, flag or other enumerable constants.

While using these primitive data types is often beneficial to programmers, this tendency is sometimes accused as *primitive obsession* [2] (Fig. 1), as it obscures the programmers’ intention and poses a threat to its safety and maintainability. One of the major benefits of using a well-defined abstract type system is its ability to check the correctness of its operations. Relying on primitive data types means that programmers are bypassing some of the necessary checks, resulting an unreliable or undefined behavior of a program. For example, in most operating systems, an arbitrary string cannot be used as a file name because a file name cannot contain certain characters (such as “/”).

While primitive obsession is known to be risky, programmers often rely on appropriate variable names and source code comments in attempt to reduce its risk by reminding themselves its intended uses and the domain specific constraints. It is known that programmers heavily rely on meaningful identifier names (type/class names, function/method names and variable/field names) to encode their intention [3].

We are interested in how programmers indicate the existence of domain specific values in source code. In this paper, we propose a novel way to identify the expressions for several predefined concepts such as a path name or calendar year. We applied our method to 26 well-known open source projects and extracted the common expressions for each concept. We then attempted to interpret the obtained results by developing a decision tree-based classifier that infers the type of concepts from a given expression. Our result indicates that there is a widely used convention to express certain conceptual types in source code. The potential applications of our technique include additional type checking, test case generation and documentation. In the example illustrated in Fig. 1, one can insert an extra check to ensure the path is correct, knowing the `path` String variable indeed specifies a file system path.

A. Contribution of This Paper

In this paper, we attempted to answer the following research questions:

- RQ.1) What kinds of high-level concepts do programmers commonly use in software projects?
- RQ.2) How do programmers express such concepts in source code?
- RQ.3) Is it possible to accurately predict such concepts from the source code appearance?

In the rest of this paper, we first define the concept of “conceptual types (c-types)” in Section III-A. We then describe how to extract conceptual types from source code in Section III-B. Section IV presents the experiment setup for obtaining conceptual type expressions and its results. In Section V, we analyzed the obtained expressions for each conceptual type by constructing a decision tree-based classifier. Finally, we discuss our findings and the threats to its validity in Section VI. The related work is described in Section II.

II. RELATED WORK

Identifying conceptual (abstract) types used in software has been an active research topic in the field of program comprehension and software maintenance. O’Callahan et al. [4] performed type inference of a given program using static data flow analysis and static point-to analysis. Their notion of a type is solely based on data flow and close to an equivalence class, in that two values can have the same type if their values can be stored into the same memory location. Guo et al. [5] took a similar approach using dynamic analysis. They also used the data flow of a program as a main source of abstract type identification. Since their method does not rely on source code, their technique could be also applied to a binary program. This line of research was further extended by Dash et al. [6]. They combined the lexical information of a program (variable names) with its data flow, forming the notion of “name flow” that was used for clustering and discovering abstract types. They also provided a facility to rewrite a program in such a way that discovered types can be automatically annotated. While it is not type inference per se, invariant detection techniques [7] can also be used for type identification, as it can discern different constraints (hence different use cases) that each variable has.

The above approaches all aimed to discover user-defined types and constraints. One of the difficulties in these problem formulations is that they are all somewhat subjective; there are a number of ways to design abstract types for a particular application. The above three approaches all used some sort of clustering technique and let the abstract types “emerge” from a program. However, it is often hard to tell if the obtained clustering was optimal for its user, as different programs have slightly different requirements for its design. Our approach is different in that our conceptual types are already well-defined by the API specification and used by many applications. While it is not directly competing with the above three, our technique can be used as a foundation of more advanced analysis.

Our approach is also related to the studies about program identifiers. The importance of names in a program code has been emphasized by many researchers and practitioners [8], [9]. Programmers generally prefer a long descriptive name than single-letter variables [10]. Poor naming can lead to misunderstanding or confusion among programmers, which eventually result in poor code quality [11]. In some software projects, inconsistent naming is actually considered as bugs (*naming bugs* [12]). Alon et al. converted source code into word embeddings [13] that correspond to a certain word in natural language [14], which can be used for identifiers.

In numerical or business applications, there are similar concepts to conceptual types that are called “dimensions”. Dimensions are typically used for expressing physical units. Jiang et al. proposed a way to add manual annotation of physical units to C programs and verify their conversion to different dimensions using predefined rules [15]. Hangal et al. used a source code revision history to check if the dimensions of each variable is consistent throughout the development [16].

III. METHODOLOGY

A. What is Conceptual Type?

Our basic idea is to use API specifications for capturing domain specific values. Well-designed API specifications usually provide a clear definition of its inputs and outputs to each function. Since programmers typically treat API functions as a black box, they need to be aware of the function parameters. More specifically, they need a precise understanding of the type of data that is being passed and how they are going to be used. Consider the following Java example:

```
String x = "foo/bar.txt";  
var f = new java.io.File(x); // x is a path name.
```

In the above snippet, the first argument of the `java.io.File` constructor is supposed to be of `String` type, according to the Java API specification. However, the programmer has to be aware that it has a more strict requirement than *just* a string because it has to be a path name. Therefore, the programmer is responsible to make sure that the value of `x` is not just a string but it meets the requirements of a valid path name (such as not containing invalid characters). In this sense, the first argument of the `File` constructor requires a more specific data type than ones that are provided by the programming language.

Conceptually, API entry points presents a clear boundary that translates primitive data types such as string to a more specific domain. In contrast to data types provided by a programming language, we call these data types a “*conceptual type*” (or “*c-type*” in short).

We identified c-types that frequently appear in the Java Standard API [17]. The principles we used in choosing these c-types are the following:

- 1) It has a clearly defined concept that is well understood by most programmers.

- 2) It is distinct enough that people do not mix up with other concepts.
- 3) It is widely used in a variety of applications.

Table I lists the 12 c-types we chose. The domain of these c-types can be divided into four different sections: file I/O, networking, GUI (Java AWT) and date/time handling. These domains are general enough that can be used in a variety of software projects.

Note that XCOORD (X coordinate) and YCOORD (Y coordinate) are treated as a separate type, as well as WIDTH and HEIGHT. These types could be merged into one, as they all represent a distance or length in a graphical device. However, programmers rarely treat these values interchangeably¹. Following the above principle 1, we consider them different c-types.

After choosing the c-types, we identified the methods that take one or more of the defined types as arguments. Table II and Fig. 2 show the number of the method arguments selected and the excerpt of these methods, respectively. In total, we selected 218 methods including overlaps. Note that some methods take multiple c-types as its arguments at once (such as WIDTH and HEIGHT).

TABLE I
CONCEPTUAL TYPES (C-TYPES)

C-Type	Actual Type	Description
PATH	String	Path name
URL	String	URL/URI
SQL	String	SQL statement
HOST	String	Host name
PORT	int	Port number
XCOORD	int	X coordinate (for GUI)
YCOORD	int	Y coordinate (for GUI)
WIDTH	int	Width (for GUI)
HEIGHT	int	Height (for GUI)
YEAR	int	Year
MONTH	int	Month
DAY	int	Day of month

TABLE II
NUMBER OF METHODS FOR EACH C-TYPE

C-Type	# Methods
PATH	14
URL	4
SQL	10
HOST	17
PORT	25
XCOORD	25
YCOORD	25
WIDTH	24
HEIGHT	24
YEAR	18
MONTH	14
DAY	18
Total	218

¹While expressions like `Point(x, x)` or `x+width*2` might be used in some programs, we can hardly imagine a GUI program where operations like `x+y` or `Point(y, x)` are meaningful.

```

• new java.io.File(PATH)
• new java.net.URI(URL)
• java.sql.Statement.execute(SQL)
• java.net.InetAddress.getByName(HOST)
• new java.net.Socket(HOST, PORT)
• new java.awt.Point(XCOORD, YCOORD)
• new java.awt.Dimension(WIDTH, HEIGHT)
• new java.util.Date(YEAR, MONTH, DAY)
• java.util.Date.setYear(YEAR)
• java.time.LocalDate.of(YEAR, MONTH, DAY)
• ...

```

Fig. 2. Excerpt of Methods used for Identifying C-Types

B. Extracting Conceptual Type Expressions

With the list of methods that define c-types, we scan the source code and identify all the calls for the selected methods or constructors. Each method call has one or more arguments that specify a predefined c-type. We then extract the expressions for each argument as a *c-type expression*.

In this paper, we use Java as our target language. First, we identify all the methods (including overloaded methods) and assign a unique identifier to each. We keep a list of method identifiers (the names and signatures) that we selected and check if each method call can match those identifiers.

In a case of virtual method call (dynamic dispatching), there are multiple method implementations that has the same signature. Note that we are only interested in the arguments of each method call; we do not need to know which method is actually invoked. When a method call can potentially invoke multiple implementations, we collect its arguments if one of its possible destinations is defined in our method list.

C. Implementation

We implemented a static analyzer for Java source code. The analyzer takes the following steps for the given set of files:

- 1) Parse all the source codes. We used Eclipse JDT [18] for the Java parser.
- 2) Enumerate all the classes and name spaces defined in the target source code. We maintain a hierarchical symbol table for registering Java packages.
- 3) Process `import` statements in each file to resolve the references to external classes.
- 4) Scan all the method signatures and assign a unique identifier to each method. For example, a method which has a signature:

```

package foo.bar;
class Config {
    int findString(String s[], int i)
}

```

can be encoded as a unique identifier:

```
foo.bar.Config.findString([LString;I)I
```

- 5) In addition to source codes, compiled Java class files and jar files are also scanned and its method signatures are collected.
- 6) Construct a symbol table that includes all the variables and field names defined in each method. The symbol table

has mappings from a variable (field) name to its data type. The symbol table is used for method resolution in the next step.

- 7) For every method or constructor call, find the most precise method that matches the calling signature.
- 8) If the callee method is one of the selected methods (i.e. its method identifier is in our list), extract the corresponding arguments that specify one of the predefined c-types.

Note that Step 3 above typically requires complete type information for imported classes. In our case, however, since we only need to identify the calls of the methods that we selected, not all the references need to be resolved. Since all the methods we chose are included in the Java Standard API, we simply ignored unresolved method calls. This allows us to process a variety of Java projects without needing its dependencies.

IV. EXPERIMENTS

We extracted c-type expressions from 26 open source projects. First we listed top 1,000 Java projects in the number of stars in GitHub. We then performed string search through their source code and selected ones that uses one or more of the Java APIs listed in Table II. We chose projects of a variety of sizes. The size of each project ranges from 1.8mLoC to 3kLoC. Table III shows the projects and their sizes.

We used a standard PC (Intel Xeon 2.2GHz, 40 core, 64G bytes memory, running Arch Linux) for running our experiment. Extracting method calls and c-type expressions for the all 26 projects took less than 2 hours in total.

TABLE III
PROJECTS AND SIZES (LoC WAS COUNTED WITH [19])

Project	Description	LoC
hadoop 3.3.1	distributed computation	1,789k
ghidra 10.0	binary analyzer	1,588k
ignite 2.10.0	distributed database	1,165k
jetty 11.0.5	web container	441k
kafka 2.7.1	stream processing	384k
tomcat 8.5.68	web server	349k
jitsi 2.10	video conference	327k
binnavi 6.1.0	binary analyzer	309k
netty 4.1.65	network library	303k
libgdx 1.10.0	game framework	272k
alluxio 2.5.0-3	data orchestration	228k
plantuml 1.2021.7	UML generator	210k
grpc 1.38.1	RPC framework	195k
jenkins 2.299	automation	177k
jmeter 5.4.1	network analyzer	145k
jedit 5.6.0	text editor	125k
gephi 0.9.2	graph visualizer	120k
zookeeper 3.7.0	distributed computation	114k
selenium 3.141.59	browser automation	91k
okhttp 4.9.1	HTTP client	36k
jhotdraw 7.0.6	graph drawing	32k
arduino 1.8.15	development environment	27k
gson 2.8.7	serialization framework	25k
websocket 1.5.2	network framework	15k
picasso 2.8	image processing	9k
jpacman	action game	3k
Total		8,480k

Table IV shows the number of extracted c-type expressions for each project. The “OTHER” column shows not an actual c-type, but the number of expressions that are passed in arguments that does not specify any predefined c-type. For example, some API method takes a path name and an extra boolean flag as arguments. Since this extra argument does not specify any predefined c-type, we count them as OTHER. The OTHER expressions are later used for training the decision tree algorithm and measuring its performance².

We collected frequently used expressions in each project. Table V shows the most frequent expressions for four c-types (PATH, URL, XCOORD and WIDTH) in each project. Constant expressions such as "localhost" are excluded. While shorter and more common expressions are relatively straightforward, a long expression with multiple operators can be complex for programmers. Table VI shows the length of expressions for each c-type, in the number of components included in each expression³. Table VII shows compound expressions that include binary operators (such as + or *).

We also obtained frequently used words for each c-type by using the word segmentation algorithm shown in Section V-B. The results are shown in Table VIII.

V. INFERRING C-TYPES BY EXPRESSIONS

In this section, we describe our attempt to develop a decision tree-based classifier that predicts the c-type from a given expression. Since the expressions obtained for each c-type contain several words that are commonly used across many projects, we expected that we could construct a relatively straightforward model (if any) to infer the c-type of a given expression.

A decision tree is a relatively simple machine learning model that is equivalent to a sequence of if-then statements. It is efficient and suitable for handling discrete values such as symbols or words. One of the major advantages of a decision tree is that it is human readable. We used a ID3 algorithm [20] to construct a decision tree.

In the rest of this section, we first describe how to decompose an expression to a set of features used for inferring conceptual types. Our classifier uses both lexical and data flow-centric information of an expression. Then we describe a word segmentation algorithm used in feature extraction. A word segmentation is needed to split identifiers that are made up with multiple words (such as `getPath`). We then show its predictive performance and an excerpt of obtained rules.

A. Converting Expression into Features

In this experiment, a c-type is specified by an argument in a method call. Each argument is a Java expression that consists of the following terms: Variable (field) accesses, method calls and constants. To use a decision tree classifier, the syntax tree of each expression needs to be converted as discrete features.

²In theory, all the arguments of all the method calls that we are not interested in should be counted as the OTHER type. For practical reasons, however, we ignored method calls that clearly have nothing to do with c-types.

³Note that field access (`a.b`) and method call (`a.b()`) are considered as two components instead of one.

TABLE IV
EXTRACTED C-TYPE EXPRESSIONS BY PROJECT

Project	LoC	PATH	URL	SQL	HOST	PORT	XCOORD	YCOORD	WIDTH	HEIGHT	YEAR	MONTH	DAY	OTHER	All
alluxio	228k	72	12	0	22	26	0	0	0	0	15	15	15	51	228
arduino	27k	39	5	0	12	9	6	6	42	42	0	0	0	1	162
binnavi	309k	42	7	1	2	1	23	23	67	67	1	0	0	1	235
gephi	120k	5	0	2	0	0	28	28	66	66	0	0	0	0	195
ghidra	1,588k	369	25	0	10	8	320	320	511	511	13	13	13	13	2,126
grpc	195k	33	16	0	68	71	0	0	0	0	0	0	0	75	263
gson	25k	0	4	0	2	0	0	0	0	0	6	6	6	7	31
hadoop	1,789k	978	634	9	288	259	0	0	0	0	2	2	2	124	2,298
ignite	1,165k	168	85	666	106	111	0	0	0	0	12	12	12	101	1,273
jedit	125k	130	19	0	3	3	50	50	112	112	0	0	0	3	482
jenkins	117k	82	28	0	6	6	1	1	1	1	102	102	102	237	669
jetty	441k	72	216	9	163	104	0	0	0	0	0	0	0	37	601
jhotdraw	32k	6	5	0	1	1	96	96	50	50	0	0	0	1	306
jitsi	327k	22	18	1	8	31	78	78	234	234	0	0	0	30	734
jmeter	145k	112	62	2	31	28	7	7	62	62	0	0	0	28	401
jpacman	3k	0	0	0	0	0	0	0	1	1	0	0	0	0	2
kafka	384k	37	1	0	85	72	0	0	0	0	14	14	14	44	281
libgdx	272k	83	7	0	4	5	7	7	36	36	0	0	0	0	185
netty	303k	38	24	0	54	130	0	0	0	0	1	1	1	117	366
okhttp	36k	2	24	0	6	7	0	0	0	0	0	0	0	3	42
picasso	9k	1	0	0	0	0	0	0	0	0	0	0	0	0	1
plantuml	210k	29	4	0	5	11	4	4	11	11	2	2	2	2	87
selenium	91k	44	66	0	15	12	1	1	0	0	1	1	1	21	163
tomcat	349k	207	64	22	38	52	0	0	10	10	0	0	0	47	450
websocket	15k	9	44	0	5	43	0	0	1	1	0	0	0	1	104
zookeeper	114k	88	4	0	126	192	0	0	1	1	0	0	0	47	459
Total	8,480k	2,668	1,374	712	1,060	1,182	621	621	1,205	1,205	169	168	168	991	12,144

Our basic idea is to focus on each identifier in an expression in the order of significance. When an expression consists of only one variable reference (such as `path`), we call this variable a *primary identifier*. When an expression consists of two references where one variable belongs to another (such as `a.b`), we choose the most significant reference (`b`) as a primary identifier as the other (`a`) as a *secondary identifier*. This strategy can be formalized by using the idea of data dependency graph (data flow graph) which has been commonly used in compiler optimization [21].

We first construct the data dependency graph of an expression by traversing each term in its syntax tree. For each term in the (sub-) expression, the rules shown in Table IX are applied recursively. The obtained graph forms a lattice structure whose node is either a variable access, method call, constant, or one of Java operators. We then traverse the dependency graph from the top and extract features at each node. Operator and constant nodes are skipped. The most significant node that is close to the top is marked as a primary identifier, and the second degree ones are marked as secondary identifiers, and so on. As we move away from the top node in the dependency graph, we obtain ternary or fourth-degree identifiers. Fig. 3 illustrates the primary and secondary identifiers that appears in a method call `new File(config.getPath(i));` The primary identifier of this expression is `getPath()`. The secondary identifier is `config` and `i`.

Note that the chain of data dependency becomes longer as we obtain a broader range of a dependency tree, i.e. the value represented at each node has a more indirect influence to the entire expression. For the sake of simplicity, we discard fourth-

degree or further identifiers.

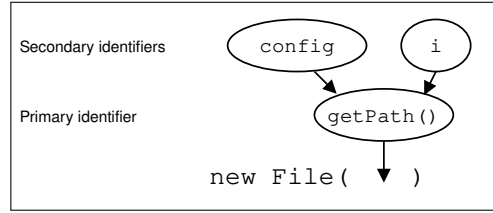


Fig. 3. Dependency Graph of “`new File(config.getPath(i))`” and Its Primary and Secondary Identifiers

B. Word Segmentation

To give the prediction model more flexibility, we treat identifiers not as a single feature but a set of features based on its tokens. For example, “`getConfigPath`” is segmented into three distinct tokens: “`get`”, “`config`” and “`path`”. Other than tokenization, the classifier does not have any prior knowledge about the natural language used in program identifiers.

We used a simple regex-based word tokenizer. For a given string, we search a longest substring that matches with $([A-Z][a-z]^+|[A-Z]^+)$ pattern. We chunk each matched substring as individual tokens. Since the extent of each match is limited to consecutive alphabets, both “`getConfigPath`” and “`get_config_path`” can be segmented to the same tokens. Each tokens is normalized to lower case letters.

TABLE V
TOP EXPRESSIONS FOR PATH, URL, XCOORD AND WIDTH C-TYPES
(CONSTANTS EXCLUDED)

PATH	Top Expressions
alluxio	path, mLocalUfsPath+ufsBase, base
arduino	path, PreferencesData.get("runtime.ide.path")
binnavi	filename, directory, pathname
gephi	System.getProperty("netbeans.user")
ghidra	getTestDirectoryPath(), path, filename
grpc	uri.getPath()
hadoop	GenericTestUtils.getRandomizedTempPath()
ignite	path, U.defaultWorkDirectory(), fileName
jedit	path, dir, directory
jenkins	System.getProperty("user.home"), war
jetty	file.getParent()
jhotdraw	prefs.get("projectFile", home)
jitsi	path, localPath
jmeter	filename, path, file
kafka	storeDirectoryPath, argument
libgdx	name, sourcePath, imagePath.replace('\\', '/')
netty	getClass().getResource("test.crt").getFile()
plantuml	filename, newName
selenium	System.getProperty("java.io.tmpdir"), logName
tomcat	pathname, path, docBase
zookeeper	path, KerberosTestUtils.getKeytabFile()
URL	Top Expressions
alluxio	journalDirectory, folder, inputDir
arduino	contribution.getUrl(), packageIndexURLString
binnavi	url, urlString
ghidra	ref, getAbsolutePath(), url.toExternalForm()
grpc	target, TARGET, oobTarget
gson	nextString, urlValue, uriValue
hadoop	uri, url, s
ignite	GridTestProperties.getProperty("p2p.uri.cls")
jedit	path, str, fileIcon
jenkins	url, site.getData().core.url, plugin.url
jetty	uri, inputUrl.toString(), s
jitsi	url, imagePath, sourceString
jmeter	url, LOCAL_HOST, requestPath
kafka	config.getString(METRICS_URL_CONFIG)
libgdx	url, URI, httpRequest.getUrl()+queryString
netty	URL, request.uri(), server
selenium	url, baseUrl, (String)raw.get("uri")
tomcat	url, location, path
websocket	uriField.getText(), uriinput.getText()
zookeeper	urlStr
XCOORD	Top Expressions
arduino	noLeft, cancelLeft
binnavi	x, m_x
gephi	currentMouseX, x, bounds.x
ghidra	x, center.x+deltaX, filterPanelBounds.x
jedit	x, event.getX(), leftButtonWidth+leftWidth
jhotdraw	evt.getX(), x, e.getX()
jitsi	x, button.getX(), dx
jmeter	graphPanel.getLocation().x, cellRect.x, x
libgdx	upButtonX, getWidth()-buttonSize.width-5, x
plantuml	e.getX()
WIDTH	Top Expressions
arduino	width, imageW, Preferences.BUTTON_WIDTH
binnavi	COLORPANEL_WIDTH, TEXTFIELD_WIDTH, width
gephi	w, constraintWidth, DEPTH
ghidra	width, center.width, filterPanelBounds.width
jedit	width, buttonSize.width, colWidth
jhotdraw	frameWidth, r.width, bounds.width
jitsi	MAX_MSG_PANE_WIDTH, WIDTH, width
jmeter	graphPanel.width
libgdx	width, buttonSize.width
plantuml	newWidth
tomcat	WIDTH

TABLE VI
EXPRESSION LENGTH (NUMBER OF COMPONENTS)

C-Type	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n \geq 7$
PATH	49.6%	22.8%	7.0%	6.3%	4.8%	2.2%	7.3%
URL	31.6%	18.5%	13.7%	13.5%	10.2%	5.8%	6.8%
SQL	47.5%	12.1%	8.7%	3.1%	4.4%	5.5%	18.8%
HOST	59.2%	11.5%	3.0%	22.1%	2.0%	0.1%	2.1%
PORT	68.4%	27.5%	2.1%	1.2%	0.3%	0.4%	0.0%
XCOORD	54.1%	24.6%	9.8%	6.4%	1.6%	2.1%	1.3%
YCOORD	52.5%	22.4%	10.1%	9.0%	2.6%	1.9%	1.4%
WIDTH	71.0%	15.2%	6.3%	2.5%	1.5%	1.8%	1.7%
HEIGHT	71.4%	15.4%	6.4%	3.1%	1.5%	1.3%	1.0%
YEAR	96.4%	2.4%	1.2%	0.0%	0.0%	0.0%	0.0%
MONTH	79.8%	19.6%	0.6%	0.0%	0.0%	0.0%	0.0%
DAY	99.4%	0.0%	0.6%	0.0%	0.0%	0.0%	0.0%

C. ID3 Algorithm

ID3 is a recursive algorithm that produces an optimal decision tree in terms of its total entropy. Our ID3 implementation is fairly straightforward. The way that the decision tree learner works is following: it scans all the input instances and searches a test that split the given instances the best. This means that a split with the minimal average entropy is chosen (Fig. 4). The average entropy of a split S is calculated as:

$$H_{avg}(S) = - \sum_{s_i \in S} \frac{s_i}{|S|} \log \frac{s_i}{|S|}$$

where s_i is the number of equivalent items in the set. The overall procedure of ID3 is shown in Fig. 5.

The algorithm starts with the most significant test, and then repeatedly splits the subtrees until it meets a certain predefined cutoff criteria; an important test tends to appear at the top of the tree, and as it descends to its branches a less significant test appears. In general, setting the cutoff threshold too small causes a tree over-fitting problem, while setting it too large makes it under-fitting. In our experiment, we found that setting the minimum threshold to 10 instances produced the best results. The more detailed mechanism is described in [20].

Once the decision tree is built, it can be treated as a sequence of if-then clauses. The classification process begins with the top node of the tree; it performs a test at each branch and decides the corresponding branch to descend. Each branch also has an associated value (prediction). When it reaches at a leaf or there is no corresponding branch, the process stops and the value associated with the current branch is returned.

Table X shows the list of ID3 features we used. The test at each branch checks if a certain word is included in one of the features. Fig. 6 shows an excerpt of the obtained rules.

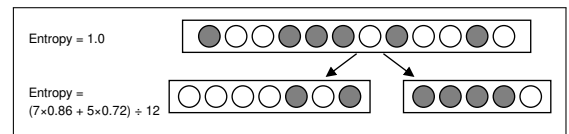


Fig. 4. Splitting Tree with Minimal Average Entropy

TABLE VII
COMPOUND EXPRESSIONS WITH OPERATORS

C-Type	Expressions
PATH	mLocalUfsPath + ufsBase selectedFile.getAbsolutePath() + PREFERENCES_FILE_EXTENSION dir.getPath() + DIR_FAILURE_SUFFIX U.defaultWorkDirectory() + separatorChar + DEFAULT_TARGET_FOLDER + separatorChar
URL	url.toExternalForm().substring(GhidraURL.PROTOCOL.length() + 1) str + KMSRESTConstants.SERVICE_VERSION + "/" newOrigin(getScheme(),getHost(),getPort()).asString() + path base + configFile
XCOORD	center.x + center.width leftButtonWidth + leftWidth evt.getX() - getInsets().left prefs.getInt(name+".x", 0)
WIDTH	Math.max(contentWidth, menuWidth) + insets.left + insets.right TITLE_X_OFFSET + titlePreferredSize.width width + insets.left + insets.right + 2 (int) (bounds.getWidth() * percent)

TABLE VIII
TOP WORDS USED IN C-TYPE EXPRESSIONS

C-Type	Top words (# Projects)
PATH	get (21), path (21), file (20)
URL	url (19), get (18), string (18)
SQL	get (6), query (5), create (3)
HOST	host (21), get (17), address (17)
PORT	port (22), get (18), local (10)
XCOORD	width (9), x (9), get (9)
YCOORD	height (9), y (9), get (8)
WIDTH	width (13), get (11), size (10)
HEIGHT	height (12), get (11), size (10)
YEAR	year (4), get (2), int (2)
MONTH	january (3), month (3), december (3)
DAY	day (3), int (2), parse (2)

```

Features = [ ... ]
MinItems = 10

def buildTree(items):
    if len(items) < MinItems:
        default = getDefaultValue(items)
        return Leaf(default)
    else:
        bestSplit = None
        for f in Features:
            split = splitItemsByFeature(items, f)
            if calcEntropy(split) < calcEntropy(bestSplit):
                bestSplit = split
        nodes = []
        for s in split:
            nodes.append(buildTree(s))
        return Tree(nodes)

```

Fig. 5. ID3 Algorithm (Python)

TABLE IX
DEPENDENCY GRAPH RULES

Expression	Dependency
# (constant)	#
A (variable access)	A
A() (method call)	A()
A.B (field access)	A → B
A.B() (instance method call)	A → B()
op A (applying a unary operator)	A → op
A op B (applying a binary operator)	A → op, B → op
B = A (assignment)	A → B

```

if "port" in PrimaryLastWords:
    if "get" in SecondaryFirstWords: ctype = PORT
    elif "host" not in PrimaryFirstWords: ctype = PORT
    ...
elif "height" in PrimaryLastWords:
    if "y" in PrimaryLastWords: ctype = YCOORD
    else: ctype = HEIGHT
elif "path" in PrimaryLastWords:
    if "host" in PrimaryLastWords:
        if PrimaryFirstWords == "host": ctype = HOST
        else: ctype = PATH
    elif "address" in PrimaryLastWords: ctype = OTHER
    ...

```

Fig. 6. Obtained Rules (Python)

TABLE X
ID3 FEATURES

Feature	Description
PrimaryFirstWords	First Words of Primary Identifiers
PrimaryLastWords	Last Words of Primary Identifiers
SecondaryFirstWords	First Words of Secondary Identifiers
SecondaryLastWords	Last Words of Secondary Identifiers

D. Classification Results

To measure the performance of our method, we conducted leave-one-project-out cross validation; For each project, we use all other 25 projects as the training data and use the one project as the test data. After repeating this project for 26 times, we took the average of the precision and recall for each project across different c-types. Table XI shows the average precision and recall as well as its F-score. The average F-score for all 12 c-types was 83%.

TABLE XI
CLASSIFICATION RESULTS FOR EACH C-TYPES

C-Type	Precision	Recall	F-score
PATH	68.9%	91.8%	78.8%
URL	61.3%	53.0%	56.8%
SQL	70.4%	80.6%	75.2%
HOST	70.0%	73.8%	71.8%
PORT	84.6%	87.5%	86.0%
XCOORD	95.7%	82.1%	88.3%
YCOORD	97.5%	79.4%	87.5%
WIDTH	92.0%	92.5%	92.2%
HEIGHT	90.4%	93.4%	91.9%
YEAR	100.0%	83.7%	91.1%
MONTH	100.0%	77.0%	87.0%
DAY	100.0%	61.1%	75.9%
Average	85.9%	79.6%	82.7%

VI. DISCUSSIONS

As shown in Section V-D, the average F-score of our classifier was about 80% for most c-types, except “URL” c-type, whose F-score was less than 60%. There are several reasons for this: First, URL expressions tend to be long and has a number of components, as shown in Table V. Most of these expressions are a concatenation of multiple strings with + operator, which is exemplified in Table V. Also, since a URL typically consists of a host name or path name, a URL expression tends to include many PATH or HOST-associated expressions as its constituents, which confuses the classifier. Indeed, this confusion is exhibited in the confusion matrix shown in Table XII⁴; a lot of URL expressions were mistaken as PATH, HOST or PORT expressions.

Now, let us go back to our research questions:

- RQ.1) What kinds of high-level concepts do programmers commonly use in software projects?
- RQ.2) How do programmers express such concepts in source code?
- RQ.3) Is it possible to accurately predict such concepts from the source code appearance?

First, we have observed that a different set of c-types appeared in different projects as shown in Table IV. Unlike general-purpose data types, the use of c-types depends on the domain of the project. This somewhat agrees with our intuition; since c-types are closer to application-specific types, its uses also depends on the application domain.

The second and third questions are related. We have seen that a decision tree-based classifier with simple features like Table X performed reasonably well for most c-types we tested. This indicates the following: there are certain conventions about how these c-types should be expressed and many programmers tend to follow them. Therefore, for conceptual types that are as common and well-defined as ours, it is relatively easy to identify them from the surface features of the source code. We think that our methodology can be extended to

⁴This matrix is obtained by applying the classifier to its own training set. Note that this is not to show the performance of the classifier. Rather, it shows the limit of its discerning ability.

a wider range of concepts in other third-party libraries and frameworks.

One of the possible ways to improve the classification accuracy is to exploit a longer data flow between method calls and other statements. In this paper, we treated individual method calls separately. However, when a method call is chained with another method call or statement, we could take advantage of this additional restriction to further refine the prediction result.

A. Threats to Validity

Here we discuss the threats to validity of our findings:

- An incomplete method list. To extract a c-type expression, we need a list of API methods that specify the corresponding c-type. We manually searched the Java Standard API documentation to find the appropriate methods for each c-type, but we might have missed some methods.
- Some c-types are rarely used in real world and we might not find enough examples. This is a classic data sparseness problem. Identifying some c-types might simply not be practical.
- Open source selection bias. Our choice of the 26 open source projects might not be representative.
- Some c-types cannot be well-defined. A primary example of this is the “URL” c-type. Technically, URL (Uniform Resource Locator) and URI (Universal Resource Identifier) are two different things [22]. URI is a broader concept which includes URL but can be used for offline entities such as book. In this paper, we treated them interchangeably because these two concepts are almost identical in the context of network applications. However, certain c-type can be more confounding and we might not be able to distinguish them in a consistent way. Another example would be “file name” and “path name”. We are yet to know how many such c-types exist.
- Potentially there are significantly more “OTHER” c-type expressions that we missed. In this paper, we assigned a hypothetical “OTHER” c-type only to arguments in certain methods. However, this should not be limited to method calls only. If we are to identify the c-type of all expressions in a program, there will be many more OTHER expressions. Training for all these OTHER expressions might confuse the classifier and end up with a much lower performance.

VII. CONCLUSION

In this paper, we set out to examine how programmers express a high-level concept such as path name or coordinates in source code. We proposed a method to identify such concepts by using standard API calls. We defined 12 c-types that are commonly used in many software projects. Each c-type can be seen as an argument for the corresponding API methods. We conducted experiments and obtained c-type expressions from 26 open source projects. We constructed a decision tree-based classifier that predicts the c-type from a given expression by combining its lexical and data flow-centric

TABLE XII
CONFUSION MATRIX OF C-TYPES

C-Type	PATH	URL	SQL	HOST	PORT	XCOORD	YCOORD	WIDTH	HEIGHT	YEAR	MONTH	DAY
PATH	2274	28	1	2	3	0	0	0	0	0	0	0
URL	82	737	2	17	54	0	0	0	0	0	0	0
SQL	2	1	287	0	4	0	0	0	0	0	0	0
HOST	7	5	0	424	2	0	0	0	0	0	0	0
PORT	2	2	2	0	755	0	0	0	1	0	0	0
XCOORD	0	0	0	0	0	478	0	6	0	0	0	0
YCOORD	0	0	0	0	0	6	486	0	7	0	0	0
WIDTH	0	0	0	0	0	3	0	597	10	0	0	0
HEIGHT	0	0	0	0	0	0	4	17	569	0	0	0
YEAR	0	0	0	0	0	0	0	0	0	18	0	0
MONTH	0	0	0	0	0	0	0	0	0	0	43	0
DAY	0	0	0	0	0	0	0	0	0	0	0	7

features. We introduced the notion of primary and secondary identifier. Our classifier achieved 83% average F-score for 12 c-types.

VIII. FUTURE WORK

There are several ways to extend our work. A straightforward extension is to support more c-types found in the Java Standard API or other third-party APIs. Since a return value of API is typically also well-defined, it is possible to extend the notion of c-type to return values.

To improve the classification performance, one can take advantage of more advanced data flow. For example, an inter-procedural data flow between different functions or bidirectional data flow between multiple statements can provide extra information to the classifier. We could also use an advanced inference algorithm such as graph neural network.

REFERENCES

- [1] J. K. Ousterhout and K. Jones, *Tcl and the Tk Toolkit (2nd ed.)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2009.
- [2] A. Shvets. (2021) Primitive obsession. [Online]. Available: <https://refactoring.guru/smells/primitive-obsession>
- [3] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ser. ICPC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 3–12. [Online]. Available: <https://doi.org/10.1109/ICPC.2006.51>
- [4] R. O'Callahan and D. Jackson, "Lackwit: A program understanding tool based on type inference," in *Proceedings of the 19th International Conference on Software Engineering*, ser. ICSE '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 338–348. [Online]. Available: <https://doi.org/10.1145/253228.253351>
- [5] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst, "Dynamic inference of abstract types," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 255–265. [Online]. Available: <https://doi.org/10.1145/1146238.1146268>
- [6] S. K. Dash, M. Allamanis, and E. T. Barr, "Refinym: Using names to refine types," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 107–117. [Online]. Available: <https://doi.org/10.1145/3236024.3236042>
- [7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 35–45, Dec. 2007.
- [8] B. W. Kernighan and R. Pike, *The Practice of Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [9] S. McConnell, *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004.
- [10] G. Beniamini, S. Gingichashvili, A. K. Orbach, and D. G. Feitelson, "Meaningful identifier names: The case of single-letter variables," in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 45–54. [Online]. Available: <https://doi.org/10.1109/ICPC.2017.18>
- [11] E. Avidan and D. G. Feitelson, "Effects of variable names on comprehension an empirical study," in *Proceedings of the 25th International Conference on Program Comprehension*, ser. ICPC '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 55–65. [Online]. Available: <https://doi.org/10.1109/ICPC.2017.27>
- [12] E. W. Høst and B. M. Ostvold, "Debugging method names," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 294–317. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_14
- [13] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *CoRR*, vol. abs/1310.4546, 2013. [Online]. Available: <http://arxiv.org/abs/1310.4546>
- [14] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019. [Online]. Available: <http://doi.acm.org/10.1145/3290353>
- [15] L. Jiang and Z. Su, "Osprey: A practical type system for validating dimensional unit correctness of c programs," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 262–271. [Online]. Available: <https://doi.org/10.1145/1134285.1134323>
- [16] S. Hangal and M. S. Lam, "Automatic dimension inference and checking for object-oriented programs," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 155–165.
- [17] (2021) Java @platform, standard edition & java development kit version 11 api specification. Oracle. [Online]. Available: <https://docs.oracle.com/en/java/javase/11/docs/api/index.html>
- [18] (2021) Eclipse java development tools (jdt). Eclipse Foundation. [Online]. Available: <https://www.eclipse.org/jdt/>
- [19] D. A. Wheeler. Sloccount. [Online]. Available: <https://dweeler.com/sloccount/>
- [20] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993.
- [21] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987. [Online]. Available: <http://doi.acm.org/10.1145/24039.24041>
- [22] T. Berners-Lee. (2021) Uniform resource identifier (uri): Generic syntax. W3C. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc3986>