srcType: A Tool for Efficient Static Type Resolution

Christian D. Newman, Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, Ohio USA
{cnewman, jmaletic}@kent.edu

Michael L. Collard
Department of Computer Science
The University of Akron
Akron, Ohio USA
collard@uakron.edu

***Abstract*—An efficient, static type resolution tool is presented. The tool is implemented on top of srcML; an XML representation of source code and abstract syntax. The approach computes the type of every identifier (i.e., function names and variable names) within the provided body of code. The result is a dictionary that can be used to lookup the type of each name. Type information includes metadata such as constness, class membership, aliasing, line number, file, and namespace. The approach is highly scalable and can generate a dictionary for Linux (13 MLOC) in less than 7 minutes. The tool is open source under a GPL license and available for download at srcML.org.**

Keywords—static type resolution, srcML, static analysis tool

I. Introduction

The goal of this tool is to statically resolve the type of each variable and function identifier within a program. Inferring types is part of our future work. A large number of software-engineering tasks rely on type information, since identifiers are such a critical part of the program. Tasks such as static analysis, slicing, reverse engineering [1], NLP for SE [2, 3], and transformation [4, 5]; all benefit from type information. A number of IDEs provide some type resolution in order to assist developers while they are programming. For example, Visual Studio and Eclipse both allow the user to check types, function signatures, etc. Other tools provide type checking at run time. Runtime type resolution is useful, but some information is lost such as templates, typedefs, etc. It is also difficult and sometimes infeasible to load a program just to try and decipher the types of various identifiers, especially if there are a large number of them to analyze.

Hence, we present an open-source tool designed to construct a symbol table and enable users to query for types and other metadata pertaining to function and variable identifiers. We will introduce the tool *srcType*, elaborate on what sorts of information it provides, discuss the API, and give a set of goals for the future of the tool.

II. Related Work

Static type checking [6] is when the type of an identifier is resolved during compilation and before runtime. After they are resolved, the compiler typically looks to see if operations performed on the types are valid for those types. If not, a compiler error is emitted. Our tool only performs the first half; computing the type of each identifier. It makes this data available for querying by the user.

Our work is somewhat related to compiler technology such as LLVM [7]. LLVM is a modular set of compiler tools that make it easier to write compilers. As such, it has tools to help in the construction of a symbol table and can be used to the same end as our tool. However, currently there does not appear to be any tools using LLVM that construct a symbol table meant for use outside of supporting the process of compilation and code generation. There are also other tools; parser generators such as ANTLR [8] can support construction of a symbol table. However, one must provide a language grammar in ANTLR before constructing the table. In short, other methods for implementing a tool like srcType are available but there are few if any open source tools to support construction of a symbol table for querying identifier metadata.

III. The srcType Tool

srcType performs static type resolution on srcML [9] documents. srcML (www.srcML.org) is a markup language that augments source code with abstract syntactic information. Using this information, srcType gathers data (using a SAX based approach) about the types and stores then in a type profile. It does not yet infer types statically; it simply scans srcML (i.e., the code) for the type provided by the developer. Every identifier in the system that represents a variable or function is given a profile. The profiles store an amalgamation of data that we discuss below. Profiles themselves are stored within a dictionary that is two maps nested within one another (see Figure 1). The first map stores function names to function profiles and the second map stores variable names to variable profiles (within that function). In order to deal with name collisions, the first dictionary's key is a combination of a function's name, the file it is in, and the line number within that file where the function is declared. Likewise, for variables, the key used in the second dictionary is a combination of the given variable's name and the line number it is declared at within the function. This ensures that every function and every identifier will be properly given a space in the map and no name collisions are possible. We now discuss each type of profile.

```
{ Function1 {var1, var2,…, varn},
  Function2 {var1, var2,…, varn},…,
  FunctionN {var1, var2,…, varn} }
```

Figure 1. Simplified architecture of two-tiered dictionary. Keys are not shown; only values.

The function profile collects data such as the function name, namespace of the function, return type, namespace of the return type, filename, line number, whether it is a method or not, whether it is a constant function, whether it returns an alias, and a dictionary of variables in the function. The structure is fairly self-explanatory. We collect data about each function in the system, all of which can be obtain via srcType's API. The last member in that list, variables, is the dictionary of variables declared within that function. There is a substantial amount of information about functions that srcType does not currently store but that we intend to implement as part of future work on the tool. This includes storing function arguments/types, data about the function's template (if it is templated), and more metadata such as if the function is inline, static, etc. The goal is the provide as much information about a the characteristics of a function as is statically computable.

Table 1 Examples of function and variable profiles.

a.	1.	void ns::foo(int x, int y){
	2.	const Object* obj = new Object(x)
	3.	}
	4.	Class test{
	5.	int bar() const{
	6.	int * x = new int[6];
	7.	}
	8.	};
a.	*FunctionProfile(foo)= {name="foo", functionNamespace="ns", returnType="void", returnTypeamespace="", filename="example.cpp", lineNumber=1, isMethod="false", isConst="false", isAliasReturn="false"}*	
b.	*FunctionProfile(bar)= {name="bar", functionNamespace="", returnType="int", returnTypeamespace="", filename="example.cpp", lineNumber=5, isMethod="true", isConst="true", isAliasReturn="false"}*	
c.	*VariableProfile(obj)= {type="Object", name="obj", namespace="", isConst="True", isAlias="True", isArray="false", isPrimitive="false", lineNumber="2"}*	
d.	*VariableProfile(x)= {type=int, name=x, namespace="", isConst=False, isAlias=True, isArray=True, isPrimitive=True, lineNumber=6}*	

The variable profile contains data such as type, name, namespace, constness, aliasing, whether it is an array, whether it is a primitive, and line number. As with functions, there is some data that we are not currently collecting that we intend to collect in the future. For variables, this includes if the variable is static, volatile, etc.

We give an example of srcType's dictionary in Table 1. At the top of the table are two functions; bar and foo. At the bottom, is a portion of the dictionary generated from the example code and includes two function and two variable profiles. The most notable difference in metadata between the function profiles is that one is a const method and the other is a free function with a namespace. For the variables, one is a non-primitive constant pointer to an object while the other is a primitive pointer to an array. We note that table is not an example of srcType's output format; only what it computes.

IV. Using srcType

The current way to use srcType is to include it as a library and use the API. The API is fairly simple and can be broken down into two steps: 1) Set the context for either function or variable; and 2) Query for a name within that context.

There are two functions for setting context. SetFunctionContext() takes a file name and a line number. The reason srcType requires this information is so that it will obtain the correct function (as opposed to an overload). The second context function is for variables: SetVariableContext(). This takes a function name and a line number for reasons similar to SetContextFunction. The requirement of having a line number and function/file name in order to perform a query can be a little restrictive since there are situations where not all information is required. For example, if every variable declaration within a function is unique, then there is no need to know which line it is on. Likewise, if a function is unique (has no overloads) then we do not need file name or line number. Hence, we are considering changing this requirement in a future version of srcType. Currently, using the API is the only way to obtain data from srcType. However, support for an output format is another of our goals. This format must contain nested data and so will likely use either JSON or XML.

Table 2. Timings of srcType

System	Size	Execution Time
Linux Kernel- 4.06	~13 MLOC	7 min
Blender-2.76	~1.5 MLOC	40 sec
Tortisesvn	~1.4 MLOC	32 sec

V. Implementation and Performance

srcType uses the srcML format and is implemented as a SAX parser. The particular implementation of SAX parser is a C++ wrapper around libxml2's SAX interface called srcSAX; it is constructed specifically to support building tools that use srcML. Since SAX parsers store no data about previously seen tags, they are very memory and computationally efficient. Table 2 presents the speed of srcType on three systems of varying sizes. The current implementation of srcType only works on C and C++ systems. However, srcML also supports C# and Java. So we intend to support C# and Java systems in the future. The only platforms supported by srcType at current are Linux and Mac.

VI. Conclusions and Future Work

We have presented a static type resolution tool that is still early in its development stages. It collects type metadata about function and variable identifiers from srcML archives so that this information can be queried through the tool's API.

While limited to simply type resolution, srcType's output is still very useful for a number of software engineering tasks. One of the biggest parts of future work is implementing full static type inference. Other goals include gathering and storing more data about functions and variables, support for additional languages, support for additional platforms, and an easy to parse output format. We also want to implement ways for srcType to handle typedefs (renaming of types) and investigate what sorts of data at the class level might be of use (i.e., keeping track of polymorphic relationships).

The project is open source and currently available for download and use from our website at srcML.org. The github page and documentation about srcType and srcML are also available from this website.

This work was supported in part by a grant from the U.S. National Science Foundation CNS 13-05292/05217.

REFERENCES

[1] N. Dragan, M. L. Collard, and J. Maletic, "Automatic identification of class stereotypes," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 2010, pp. 1-10.

[2] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of NL-queries for software maintenance and reuse," presented at the Proceedings of the 31st International Conference on Software Engineering, 2009.

[3] R. S. Alsuhaibani, C. D. Newman, M. L. Collard, and J. I. Maletic, "Heuristic-based part-of-speech tagging of source code identifiers and comments," in *Mining Unstructured Data (MUD), 2015 IEEE 5th Workshop on*, 2015, pp. 1-6.

[4] N. Meng, M. Kim, and K. S. McKinley, "LASE: locating and applying systematic edits by learning from examples," presented at the Proceedings of the 2013 International Conference on Software Engineering, San Francisco, CA, USA, 2013.

[5] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: generating program transformations from an example," *SIGPLAN Not.,* vol. 46, pp. 329-342, 2011.

[6] R. W. Sebesta, *Concepts of Programming Languages*: Pearson, 2012.

[7] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," presented at the Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, Palo Alto, California, 2004.

[8] T. Parr, *The Definitive ANTLR 4 Reference*: Pragmatic Bookshelf, 2013.

[9] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration," presented at the Proceedings of the 2013 IEEE International Conference on Software Maintenance, 2013.

[1]