Bash2py: A Bash to Python Translator

Ian Davis, Mike Wexler, Cheng Zhang, Ric Holt
David Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{ijdavis, mwexler, c16zhang, holt}@uwaterloo.ca

***Abstract*—Shell scripting is the primary way for programmers to interact at a high level with operating systems. For decades bash shell scripts have thus been used to accomplish various tasks. But Bash has a counter-intuitive syntax that is not well understood by modern programmers and is no longer adequately supported, making it now difficult to maintain. Bash also suffers from poor performance, memory leakage problems, and limited functionality which make continued dependence on it problematic. At the request of our industrial partner, we therefore developed a source-to-source translator, bash2py, which converts bash scripts into Python. Bash2py leverages the open source bash code, and the internal parser employed by Bash to parse any bash script. However, bash2py re-implements the variable expansion that occurs in Bash to better generate correct Python code. Bash2py correctly converts most Bash into Python, but does require human intervention to handle constructs that cannot easily be automatically translated. In our experiments on real-world open source bash scripts bash2py successfully translates 90% of the code. Feedback from our industrial partner confirms the usefulness of bash2py in practice.**

Keywords—Bash; Python; scripting; open-source; translation

I. INTRODUCTION

Shell scripts are frequently used in system administration, software development, and maintenance. Shell scripts can organize long and repetitive commands in a structural way, enabling the automation of various tasks, such as creating user accounts, downloading and installing software packages, initializing environment variables, etc. Shell scripting was ranked the 7th most popular programming style in 2013 [1].

Despite its prevalence, Bash (Bourne-Again SHell) [2] is problematic. Its syntax is often counter-intuitive. For example, programmers must pay special attention to the treatment of spaces and brackets to read or write bash shell scripts correctly. Bash does not invite reuse since it lacks modularity and object-oriented facilities. It also lacks the host of libraries available to developers using more modern programming languages (e.g., Python), so functionality is often achieved by gluing specific utility programs together. Trouble-shooting and diagnosis of runtime errors can be unnecessarily complicated. These issues make it a daunting task to maintain a large number of bash shell scripts, especially when most of these scripts were written by developers now long gone.

One solution to this bash maintainability problem is to replace bash shell scripts with scripts written in newer, more maintainable, and more powerful, scripting languages. However, manually translating many poorly understood bash scripts involves a huge investment in time and effort, and invites disaster. In this paper, we describe a code translator, *bash2py,* which automatically converts bash scripts into Python. This is significant since Bash is now the default shell of GNU/Linux as well as many other popular Linux distributions, such as Ubuntu, Fedora, etc.

Our industrial partner, Owl Computing Technologies Inc. [3], chose to translate their bash scripts into Python because of its increasing popularity; its continued support and development; its functionality and improved security; its clarity as a language; its superior run time performance when compared to Bash; and its suitability for use within web based applications and graphical user interfaces. Python comes with a large number of supporting modules and features, is extensible, and has good tools for validating syntax (e.g., Pylint) and clearer runtime diagnostics when errors occur.

Bash2py leverages the open-source parsing component of bash 4.2 to obtain the tree-like intermediate representation of each bash command. Using this intermediate representation, bash2py combines word expansions of bash and context-aware string generation to produce Python output. During the translation process, indentation is treated with care, as it is especially important in Python. Comments are also preserved and associated with their corresponding Python code.

To evaluate the usefulness of bash2py, we ran it on six real-world open source projects written in Bash. We found that bash2py can translate 90% of the bash code, leaving only minor issues that can be easily fixed manually. Owl CTI has been using bash2py to migrate all their bash scripts to Python, and their feedback is quite positive: Almost all the content of their scripts can be translated automatically and effortlessly. The resulting python scripts are then much easier to maintain.

Bash2py is available to everyone as an open-source C tool [4]. It should be simple to compile and run on any system that already supports the compilation and execution of Bash. Both Bash and Bash2py are free software, distributed under the terms of version 3 of the [GNU] General Public License as published by the Free Software Foundation.

II. TYPICAL USAGE

Bash2py is typically used in migrating legacy bash scripts to Python, with only the latter when correct subsequently being maintained. But it can also be used to generate a side-by-side comparison of Bash and Python syntax, as a reference that can be helpful in maintaining the original bash code.

Bash2py is an easy-to-use command line tool. The user specifies the input bash script file or the directory containing one or more input files as the command line argument. The user can use the following options to invoke bash2py.

bash2py [-h] [-f|-d] <filename>

The *–f* option indicates that the provided filename is a regular file containing a bash script and the *–d* option that it is a directory that should be recursively scanned for bash scripts. When neither option is specified it is inferred from the provided filename. The *–h* option indicates that comparative html output should be generated, rather than a python script.

III. DESIGN AND IMPLEMENTATION

The overall architecture of *bash2py* which is derived from that of Bash is shown in Figure 1(b). As shown in Figure 1(a), Bash consists of three main components: lexical analysis and parsing; text expansion; and command execution. The input may come from an interactive user interface (e.g., console) or from a bash script file. The first component parses each command, which can be either simple or compound, into internal data structures. Then the second component performs a series of expansions and variable substitutions on the parsed command, by following a set of sophisticated rules. Each resulting command is then interpreted; with at end of run Bash returning an exit status code.

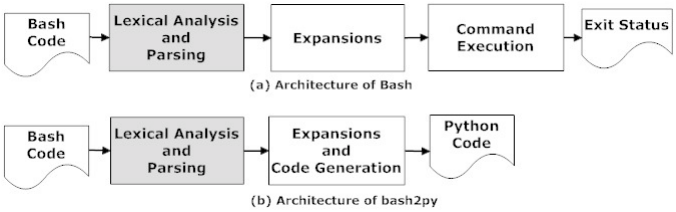

Fig. 1. Architecture of bash and bash2py.

A. Parsing of bash code

Bash2py reuses the bash 4.2 lexical analysis and parsing, to obtain the same data structures used in Bash, and to produce the same syntactic error messages. This ensures that bash2py can parse every bash construct. Internally *bison* [5] parses commands. While it would be both expensive and difficult to create our own bash parser, a customized parser might provide richer information which would then simplify subsequent code generation. The original bash component parses only the high level constructs within the bash language, and leaves much of the syntactic interpretation of the detailed input to ad-hoc logic. In addition, complicated lexical analysis is required simply to correctly tokenize the bash input stream and to thus assist the parser [6]. This logic has grown and evolved over time, is now something of a mess, and is inherently less informative than a more complete bash grammar would be.

B. Combining expansions with code generation

Before executing a command, Bash performs sophisticated potentially nested replacement of embedded constructs within a command. Bash recognizes and implements brace expansion, tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, word splitting, and filename expansion [7]. These expansions permit the values of variables to be recovered, the value of computations to be determined, and the output of external commands to be used. They may also result in side-effects, such as the declaration of an undefined variable and the assignment of a value to it.

Expansions can also simplify bash scripting. For example, brace expansion makes *pre{a,b,c}post* into three strings, *preapost*, *prebpost*, and *precpost*. But convoluted expansion rules often make bash code hard to understand. They also complicate the process of code generation within bash2py, since often there are no obvious equivalent Python constructs.

Bash2py quotes unquoted text strings, and makes explicit the implicit concatenation of string fragments permitted in Bash, but not in Python. Rather than expand variables, expressions and command invocations to their resulting values, bash2py expands these constructs to their nearest Python equivalents. Since bash2py is a code translator it does not execute commands. In a future release of bash2py all of the extant bash code used only during bash command execution will be removed from our distribution, making our code smaller, simpler, and subsequently incapable of erroneously executing this redundant bash code.

Bash2py does not translate all bash expansions correctly, but it does correctly translate most of the simpler expansions. It must keep track of variables used within functions, and cases where these variables are declared to be local, so that it can correctly preserve bash semantics, by declaring all non-local variables used within a bash function to be global in Python. Expansions that produce side effects are expressed in Python as statements that implement these side effects, but often such Python statements may not be legally embedded within surrounding text, as is permitted in bash.

In addition to bash expansion mechanisms, bash2py indiscriminately replaces many common keywords that occur in bash with their Python equivalents whenever they are seen. For example, the bash operators *-gt*, *-eq*, and *!*, are translated to >, ==, and *not*, respectively. This can result in erroneous output.

In Bash most constructs return a value which may then be tested. Often this is used with lazy and/or evaluation to concisely express when actions should be performed. Such programming nuances cannot readily be translated into Python, which expects conditions to be braced by if's and else's.

C. Dealing with indentation and comments

Indentation is of particular importance in Python, as the depth of indentation is used to determine grouping of statements. Contrariwise, indentation is only used for

readability in Bash, so it provides little information on how to generate indentation in the corresponding Python code. To address this issue, bash2py systematically calculates the amount of indentation at each relevant position in bash code, and inserts the indentation accordingly during code generation.

Actual indentation is achieved by using a global variable to indicate the desired level of indentation, and low level print routines which transparently insert the desired indentation, each time text is emitted that immediately follows a new line character. The high level translation software is otherwise oblivious to the need to indent.

Unlike indentation, code comments have no impact on program behavior, but they often contain information useful to a programmer. Unfortunately, the bash parser simply discards comments during the tokenization process. To reintroduce comments into the output stream, we instrumented the bash source to accurately record where within the input each bash statement and comment started. Comments were then temporarily buffered as seen, but not emitted until processing a bash statement that logically occurred after them. They were then emitted in the order seen. This preserved the sequential ordering of comments and commands within the output.

D. Main bash structures translated

Within the current design and implementation, bash2py focuses on translating the most frequently used language structures in bash, which can be summarized as follows:

- Simple single statement commands.
- Control constructs, including conditional constructs such as if, select, case, and loops such as for, while and until.
- Grouped commands grouped with () or {}.
- Commands connected with |, &, ||, &&, etc.
- Function declarations.
- Function invocations.
- Extensions for input/output redirection, etc.

For each simple command, bash2py first checks whether the command is a recognized keyword having a known translation to Python. For example, the built-in command *echo* can be translated to *print* in Python while the built-in command *cd* which changes the current working directory in Bash, is translated as *os.chdir()* in Python.

When the command is not recognized, it can be a request to invoke the same named internal function, or to invoke an external operating system command. We distinguish between these two possibilities by remembering the names of all previously declared functions within a script.

External commands are invoked using subprocess.call(). When the command involves complex shell constructs, we perform a shell invocation, but for invocations involving only a program name and arguments we encode these as an array, since this is less vulnerable to certain well known shell exploits. For example the three bash commands:

```
cat -b "abc" > def
cat -b "abc"
cat <<EOF
Usage: $0 [options]
-h help
EOF
```

are translated as:

```
_rc=subprocess.call("cat"+" "+"-b"+" "+
"abc",shell=True,stdout=file('def','wb'))

_rc=subprocess.call(["cat","-b","abc"])

_rc = subprocess.Popen( "cat",
shell=True, stdin=subprocess.PIPE)
_rc.communicate("Usage" +str(__file__)+
": [option]\n-h help\n")
```

IV. EXPERIMENTS AND INDUSTRIAL EXPERIENCE

During our consultation with Owl, we were given access to a project that uses a heavy amount of shell scripting for its installation process on the client machine and for monitoring client-server interfaces. This shell code was written primarily in Bash, and the rest using the Korn-shell. The project we worked with had a total of 87 bash files, which contained a total of 33,109 lines of bash code. This yields an average of 380 lines per file. The median number of lines is 210, and the maximum number of lines found in a file was 2310. The standard deviation was 437. One of the software developers at OWL used bash2py to translate 640 lines of bash code. By manual inspection, he found that only 26 of these lines contained errors. Thus, 96% of the translated lines were translated correctly.

To further evaluate bash2py, we selected six open-source bash projects from GitHub (links are available at [4]). All these projects are written only in Bash, and they implement diverse functions, using a wide range of bash language features. We selected projects of moderate size, since it was feasible to manually validate only small projects. In Table 1, the column #Bash shows the size of subject programs, and the columns #Python and #Rejected show the size of the translated Python code and the lines of code rejected by pylint, respectively. All the numbers are measured in non-comment non- blank lines of code. The last column shows the success rate of the translation, which is equal to (#Python-#Rejected)/#Python. Because the thorough functional specifications are unknown for the subjects, we checked the output Python manually with the help of pylint [8], a code checker for Python. The output for these experiments is available [4]. As shown in Table 1, bash2py translates 687 out

of 767 lines of code correctly, with a success rate of 90%.

Table 1. Evaluation of bash2py on open-source projects

Program	#Bash	#Python	#Rejected	Success
assert	113	119	18	85%
bash2048	245	250	10	96%
bashmarks	67	71	6	92%
bashtime	121	83	7	92%
bashttpd	124	93	23	75%
JSON	173	151	16	89%
Total	843	767	80	90%

IV. RELATED WORK

The idea of converting bash code to Python has been discussed among programming communities [9]. In [10], Delaney lists several disadvantages of shell scripting and discusses the benefits of replacing Bash with Python. The article also exemplifies how to use Python, in place of Bash, to fulfill tasks, such as data processing and sending emails. Gift [11] gives a step-by-step tutorial on how to write Python code based on relevant knowledge of Bash. He states that Python is easier to program with and usually outperforms Bash as a scripting language. The development of bash2py is largely inspired by these discussions and Owl's desire to switch to Python.

There are many tools for source-to-source code transformation. The ROSE compiler infrastructure [12] can be used to translate and analyze several popular programming languages, including C, C++, Fortran, Java, Python, PHP, etc. The 2to3 tool automatically migrates code in Python 2 to Python 3 [13]. The Google Webkit Tool (GWT) [14] allows developers to use specific Java APIs to create web applications which are automatically translated into JavaScript programs. Among these various kinds of tools, a common approach is to first parse/transform the source program into some intermediate representation (IR), and then generate the target output from the IR. Bash2py takes a similar approach, in which the IR is the internal bash COMMAND data structure. To the best of our knowledge, bash2py is the first publicly available automatic translator from Bash to Python.

V. LIMITATIONS AND FUTURE WORK

Bash2py was developed with the specific goal of dramatically reducing timeframes needed to port a substantial body of industrial code written in Bash to Python. It assists in this translation process by correctly translating most of the bash control structures presented to it, and in migrating from loose string representations in Bash, to the much stricter syntax required by Python. Bash2py does not presume to be capable of accurately translating everything presented to it. It does not translate correctly all the nuances of parameter substitution available to a bash programmer; it does not recognize all of the mechanisms available in Bash for input and output redirection; and it cannot adequately address significant semantic differences between Bash and Python.

In Bash variable typing is inferred at run time, and so not known to our translator. We therefore presume that all variables may contain strings, and coerce them to strings when this is in doubt. We recognize that for strictly numerical variables considerable ugliness and needless overhead results. In Bash a string variable containing newline characters may transparently be treated as an array having one entry per line. Manual intervention to translate a string to an array is required to achieve this same effect in Python.

Our future work will be to continue to support, and improve Bash2py. Bash2py currently employs a somewhat ad-hoc sequence of translations to transform bash strings with embedded parameters into Python. Better results might be obtained if we generated Python strings by modifying the existing low level runtime code that expands strings within Bash. While obviously important, there are currently virtually no diagnostics advising users of code known to have not been correctly translated. This needs to be addressed. Substantially more testing is required to discover and rectify cases where our existing translation is flawed, much of which we hope can be achieved by our industrial partner, and the wider community.

ACKNOWLEDGMENT

This work was inspired, funded and supported by Ron Mraz, the president of Owl CTI. Theresa Weber as Owl's project manager was enormously helpful in being our champion at Owl CTI. Robb Zucker and Adam Laughlin both assisted in validating the usefulness of this tool to Owl.

REFERENCES

[1] *Programming Language Popularity*. http://www.langpop.com

[2] C. Ramey and B. Fox, *Bash Reference Manual*. Network Theory Limited, 2003.

[3] Owl Computing Technologies Inc. *Securing your networks from cyber threats*. http://www.owlcti.com

[4] SWAG Software Architecture Group. *Bash to Python script translator*. http://www.swag.uwaterloo.ca/bash2py.

[5] *Bison GNU parser generator*. http://www.gmu.org/software/bison

[6] C. Ramey. *The bourne-again shell The architecture of open-source applications*. http://aosabook.org/en/bash.html 2011.

[7] *Bourne-Again Shell manual*. http://www.gnu.org/software/bash/manual.

[8] Pylint Home Page. http://www.pylint.org. 2014

[9] C. Jefferson. *Can I use Python as a Bash replacement?* http://stackoverflow.com/questions/209470/can-i-use-python-as-a-bash-replacement

[10] R. Delaney. *Python scripts as a replacement for bash utility scripts*. Linux Journal, vol 2012 no 223. November 2012.

[11] N. Gift. *Python for bash scripters: A well-kept secret*. Red Hat Magazine, http://magazine.redhat.com/2008/02/07/python-for-bash-scripters-a-well-kept-secret 7 February 2008.

[12] *ROSE compiler infrastructure*. http://rosecompiler.org 2014

[13] *2to3 – Automated Python 2 to 3 code translation*. https://docs.python.org/2/library/2to3.html 2014

[14] *Google Web Toolkit*. http://www.gwtproject.org