Type-directed decompilation of shell scripts

Spencer Baugh
University of Carcosa
Carcaso, Hyades
first.last@gmail.com

Dougal Pugson
Pugson's C++ Crypt LLC
New York, USA
dougalpugson@gmail.com

Abstract

Maintenance shell programmers are often faced with inscrutable shell scripts without human-readable source code. We apply techniques pioneered by the type-directed partial evaluation community to create a decompiler which can take an executable shell script and recover its original source code. This technique has surprising generality, and our decompiler can also be used as a pretty-printer, or in general, as a compiler from any language into shell.

CCS Concepts • **Software and its engineering → Scripting languages**; **Compilers**; **Software maintenance tools**; • **Theory of computation → Type theory**.

Keywords Partial evaluation, type systems, shell scripting

ACM Reference Format:
Spencer Baugh and Dougal Pugson. 2020. Type-directed decompilation of shell scripts. In *Proceedings of SIGBOVIK (SIGBOVIK 2020).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.475/123_4

1 Introduction

Among its many beneficial properties, shell has a unique feature: The language automatically compiles itself as it is written. Immediately after a shell script is typed into a computer and saved into a file, it is transformed into a compiled form which is unreadable by humans. [1] Such a script can be immediately used for all your most important financial data, medical procedures, etc., while you are safe in the knowledge that no-one can read your important proprietary shell scripts.

Unfortunately, sometimes such scripts might exhibit minor bugs. Since these executable files that are leftover from the process of shell scripting are unreadable garbage, maintenance programmers are often forced to rewrite from scratch. Sometimes, very brave programmers will try open one of these old, buggy "shell script files", but generally they are instantly turned to stone upon seeing the inscrutable contents. And since shell is so convenient, typically the original programmer is long gone - probably retired early.

Our contribution in this paper is a decompiler for these files, which is able to recover the original source code. This source code can then be viewed by the maintenance programmer, to help them reimplement the script in their non-shell language of choice.

As outlined in our previous work, [1], the Unix shell is closely historically related to functional programming. Thus it should be no surprise that we are able to transplant techniques from the functional programming community, to the shell, in the tradition of [4].

In this case, we use partial evaluation techniques to achieve our goal. As described in [3], as citing [2], as blowing the minds of undergrads everywhere, sufficiently general partial evaluation techniques can be applied to "reify" a compiled program and recover the program source code, under the constraints that the program 1. has a type and 2. terminates. Pretty easy constraints, I think we can make that happen for bash!

As a more approachable introduction (note that "more approachable" doesn't mean "approachable"), these techniques can be compared to tagless-final-style techniques. Decompilation of a typed, compiled program is essentially identical to pretty-printing of a tagless-final-style term. The magic is that any typed, closed program can be treated as a TFS term.

Our paper is organized in some number of sections. Section 1 contains an introduction and a description of the organization of the paper. Section 2 contains a exploration of the requirements for applying these techniques, and establishes their firm grounding in theory. Section 3 demonstrates several applications. Section 4 gives an overview of the implementation. Section 5 concludes the paper, discusses future work, and affirms that this was the right thing to do.

2 Theory

As mentioned above and in [2], we have two requirements to apply type-directed partial evaluation: The program must be typed and must terminate.

2.1 Typed

What is the type of a shell script? Well, a shell script takes the PATH environment variable, and runs commands (identified by strings) out of PATH, each of which has some side effects.

[1]Use of the authors' previous work on delimited continuations in bash, [1], seems to accelerate this process.

SIGBOVIK 2020, April 2020, Pittsburgh, PA USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

A shell script doesn't return anything, it just has side effects, so let's say that its output type is unit.

So that means a shell script has type $path- > unit$, where $path$ is $(str, [str])- > unit$.

Then in total, a shell script has type $((str, [str])- > unit)- > unit$.

We can confirm this is correct by applying double negation elimination [2] which shows us that shell scripts have type $(str, [str])$. This is correct because a shell script is indeed a bunch of strings.

Let's be a little more specific with our type, though, and model each executable as a function. So then the type of PATH is $(str- > ([str]- > unit))$, and the type of a shell script is then $(str- > ([str]- > unit))- > unit$

For now, just think of a shell script as taking PATH and running commands out of it.

2.2 Termination

Termination, on the other hand, is a much harder problem than assigning a strict static type to shell scripts. This is because of the presence of the dreaded D-wait in Unix. A process can get into an uninterruptible state when making a filesystem request, and just hang forever, ignoring all signals, including SIGKILL. This is really annoying for people developing filesystems, which we had to do for this paper, so we want to complain about it here.

Nevertheless, if the program hangs, this issue can be solved by simply mashing Ctrl-C. Even D-wait can be solved by throwing the computer out the window (assuming a sufficiently portable computer, and great enough height for it to be destroyed on impact).

So termination is ultimately not a problem either.

2.3 Background

In brief, the principle of the technique we will be applying is this: A closed, abstract function, which takes in other functions and combines them through application in some way to eventually return a result, can be passed functions which, instead of performing actual operations and returning real results, take ASTs and return ASTs.

For example, a parameter with type $(a, a)- > a$, which might normally be addition of two integers or something, can be passed at the specific type $(ast, ast)- > ast$, and be implemented as $\lambda x.\lambda y.Plus(x, y)$ where $Plus$ is some datatype constructor.

A shell script's single argument (in our model) is the PATH environment variable. [3] We will pass in a PATH which, when an executable name is looked up in it, returns the executables (functions) of our choice. These executables will in turn, when executed, construct an AST instead of actually doing anything.

[2] DNE is an axiom because this paper is unintuitive.

[3] This has sufficiently abstract types, since everything is a string, so we don't know what anything is.

3 Applications

Let's demonstrate our tool before getting to the actually interesting part.

3.1 Decompiling bash

Suppose we save the following shell script to a file and mark it executable, which instantaneously makes it unreadable.

```
ls ; which ls
stat /
foo|bar
```

Nevertheless our decompiler can run on the script and produces the following output:

```
ls
which ls
stat /
foo | bar
```

As you can see, our decompiler even pretty-prints the shell script.

3.2 Decompiling arbitrary executables

It also works on C programs, and in general, arbitrary executables. We can compile the following normal C program, and run our decompiler on it.

```
int main() {
  int rc;
  rc = fork();
  if (rc == 0) {
    execlp("foo",
           "foo", "bar", "baz", NULL);
  } else { wait(NULL); }
  rc = fork();
  if (rc == 0) {
    execlp("whatever",
           "whatever", "quux", NULL);
  } else { wait(NULL); }
  return 0;
}
```

And we get the following shell script out:

```
foo bar baz
whatever quux
```

Useful!

3.3 Optimizing decompiler

Our decompiler is so advanced that it in fact transparently applies optimizations in the process of decompilation. Consider the following C program:

```
int main() {
  printf("hello␣world\n");
}
```

This program decompiles to the following shell script:

Our decompiler correctly recognizes that this program, since it doesn't execute any other programs from the filesystem, is in fact utterly worthless, and optimizes it away to nothing.

4 Implementation

Much to our surprise, we actually implemented this.

We have implemented a filesystem (using FUSE), which pretends to have any possible executable you want. We point the shell script at this filesystem using PATH, and each time the shell script goes to run a command, it instead runs a stub under our control. Using some real technology we developed earlier and just thought it would be funny to use for this, this stub connects back to the filesystem server, where our decompiler is able to query its argv, stat its stdin/out/err, and tell it to exit with a specific exit code.

To mount the filesystem without requiring privileges or setuid executables, we use a user namespace and mount namespace, and run the script inside those namespaces.

After the shell script finishes execution, we reconstruct its source code from the trace of executed commands using a highly advanced for loop.

Note that this doesn't use "LD_PRELOAD" or strace, so it can even be used on statically linked, setuid shell scripts. There are lots of those!

The code is on Github at https://github.com/catern/rsyscall/tree/master/research/sigbovik2020.

5 Conclusion

5.1 Future work

5.1.1 Support for niche shell features

There are some niche, minor features of the shell language which are not supported by our decompiler, such as "if" and "while". As any true shell programmer uses "xargs" instead, which our framework would decompile just fine, this isn't a problem.

Nevertheless it might be nice to figure out some ridiculous hack that would allow such features (and shell builtins in general) to be visible to our decompiler.

Maybe we could execute the shell script multiple times, returning different exit codes from commands different times, and thereby get a collection of traces through the control flow graph, which we could then piece back together. But this is beginning to sound like real work.

5.2 Conclusion conclusion

In conclusion, we hope that this tool proves useful for maintenance shell programmers, who will finally have a way to read those shell scripts that they always complain are unreadable. Hopefully this will increase their productivity, massively increase global GDP, and cause my 401K to recover all the value it lost due to the coronavirus.

References

[1] Baugh, S. bashcc: Multi-prompt one-shot delimited continuations for bash. In *Proceedings of SIGBOVIK 2018* (Pittsburgh, Pennsylvania, 2018), Association for Computational Heresy, pp. 161–164.

[2] Danvy, O. Type-directed partial evaluation. In *Partial Evaluation* (Berlin, Heidelberg, 1999), J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, Eds., Springer Berlin Heidelberg, pp. 367–411.

[3] Kiselyov, O. *Typed Tagless Final Interpreters.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 130–174.

[4] Shelley, M. W. *Frankenstein: or The modern prometheus.* Lackington, Hughes, Harding, Mavor, and Jones, 1818.