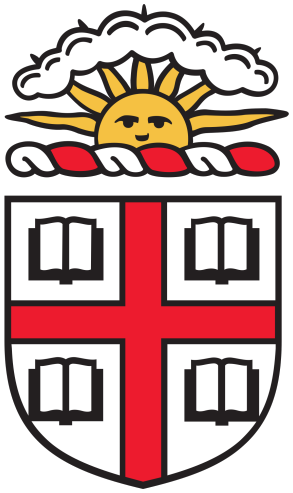
Bash-Compliant PaSh



Seth Sabar

Department of Computer Science

Brown University

Supervisor

Nikos Vasilakis Ph.D.

In partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science

May 26, 2024

Acknowledgements

I would first like to thank Nikos Vasilakis, my thesis advisor, for supporting me throughout the research process. Professor Vasilakis was incredibly supportive, especially with many of the logistical aspects of completing a thesis. He also generously allowed me to join his research group in my junior year, where I gained the foundational knowledge to complete this work.

I would also like to extend a major thank you to Michael Greenberg, a computer science Professor at The Stevens Institute of Technology. Professor Greenberg acted as my primary technical advisor, helping me overcome challenges every step of the way. His project *libdash* was the framework from which I created *libbash*, a major component of this project.

I would also like to thank Konstantinos Kallas, a PhD student at The University of Pennsylvania. Dr. Kallas is a major contributor to PaSh and was always willing to give me technical help on my work.

Finally, I'd like to thank the entirety of the PaSh research group. It's truly inspiring to see how a group of undergraduate students, master's students, PhD students, and professors unite to build an incredible body of research and tools.

Abstract

Shell scripting is one of the most popular programming languages in the world, listed 9th on Github in 2023. *PaSh*, a recent shell-script optimization project hosted by the Linux Foundation, allows shell programs to reap multiprocessing benefits fully automatically. Unfortunately, *PaSh* only works on scripts written in a subset called POSIX—a standard created to define a subset of features that all shell interpreters should have. In reality, however, users do not write shell scripts to match the POSIX standard, they write scripts for their interpreter of choice. *Bash*, or the Bourne Again Shell, is the most popular shell interpreter, and the default interpreter for the most popular Linux distributions.

This dissertation outlines how *PaSh* was expanded to include compatibility with all *bash* scripts. To complete this work, we implemented a *bash* parser, integrated the *bash* Abstract Syntax Tree (AST) into a generic *shell* script AST interface, and integrated this work into the *PaSh* tool. This dissertation demonstrates that the addition of *bash*-compliance to *PaSh* continues to enjoy the same speed-ups and correctness seen with POSIX-compliant *PaSh*.

Contents

1 Introduction **1**

2 Background **4**

2.1 Shell and POSIX 4

2.2 Bash 6

2.3 PaSh and Parallelization 7

3 Design **10**

3.1 libbash 11

3.1.1 Bash Source Code 12

3.1.2 Python interface 19

3.2 Shasta Integration 21

3.2.1 Libbash to Shasta 21

3.2.2 Adding Shasta Nodes 22

3.2.3 Printing Shasta Nodes 26

3.3 PaSh Integration 28

3.3.1 Preprocessing 28

3.3.2 Compilation 29

4 Evaluation **30**

4.1 Correctness 30

4.1.1 libbash 30

4.1.2 shasta 32

4.1.3 PaSh 32

4.2 Efficiency 33

5 Limitations **38**

5.1 Bugs . . . 38

5.1.1 Bash Command Printing . . . 39

5.1.2 Bash Parsing . . . 40

5.1.3 Setting IFS . . . 41

5.2 Inherent Limitations . . . 41

5.2.1 Commands that Change Parsing . . . 42

5.2.2 Bash Variables . . . 43

6 Future Work **44**

6.1 Native Expansions in Libbash to Shasta . . . 44

6.2 Handling Commands That Change How Parsing Works . . . 45

6.3 PaSh Compliance with Other Shells . . . 46

7 Conclusions **47**

References **48**

8 Appendix **51**

8.1 Known Limitations Table . . . 51

Chapter 1

Introduction

Despite being around for almost 50 years, *shell* continues to be one of the most popular programming languages in the world, ranking 9^{th} in languages used in 2023, and seeing an almost 20% growth in contributors according to Octoberverse [1]. The *shell* continues to be among the most popular languages in the world for a few reasons. For one, *shell* scripts are portable—nearly every modern operating system, including MacOS, Windows, and Linux has a built-in *shell* interpreter and terminal. Moreover, in the *shell* programming language, it is much easier to interact directly with the host's file system and files—operations that take several lines of code in other languages can be done in a few characters with the *shell*. Finally, the *shell* programming language has powerful data processing tools that allow developers to write complex data pipelines in just a few lines of code.

The *shell*, however, is not exactly one programming language but instead refers to a set of similar, but not identical programming languages, that a user usually interacts with via a terminal—a text-based user interface to execute commands and view results. When *shell* interpreters were relatively new, several different implementations emerged, and some developers were concerned that these implementations could diverge too far from each other. Thus, the POSIX standard was developed to provide a list of rules that all *shell* interpreters must follow [2, 3]. This standard is followed closely today by nearly all popular *shell* interpreters, however, many *shell* variants have added additional features beyond those required by the POSIX specification. Though there are several different popular *shell* interpreters today, *bash* is recognized as the most popular [4, 5].

Despite its prevalence in data processing, *bash* and other popular *shell* variants often miss opportunities to parallelize scripts, especially with respect to data-parallel script execution. In many modern development environments, machines have up to 64 processors or even, meaning that in many cases computation that could be done in a few hours can take several days. Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković worked to solve this issue with their release of *PaSh* [6], a system for parallelizing POSIX *shell* scripts. A user inputs their *shell* script to *PaSh*, which analyzes it for parallelization opportunities, outputs a script with this parallelization built-in and runs it. *PaSh* proved to be successful, getting speedups from 1.92 to 17.42 times on a series of benchmarks [6]. *PaSh* has also proven to be useful for users. The PaSh GitHub repository has 529 stars on GitHub at the time of writing [7]. The Linux Foundation also hosts PaSh as of September 2021, citing industrial use cases as a reason for hosting [8].

Despite its popularity, *PaSh* currently only supports POSIX-compliant scripts—that is, scripts with only features that all of the *shell* variants are expected to have. In some cases this is acceptable, however, in practice, users often write scripts for the *shell* implementation they use. Very often, especially in Linux environments (the only operating system *PaSh* currently only supports), *bash* is the *shell* implementation of choice. *Bash* is the default *shell* interpreter on most major Linux distributions. This puts users of *PaSh* in a difficult spot. The purpose of *PaSh* is to run *shell* scripts much faster than they would run otherwise. However, much of this convenience is lost if users have to avoid using *Bash*-specific features (commonly referred to as bashisms).

This dissertation aims to solve this issue by adding *bash*-compliance to *PaSh*. That is, a *PaSh* user should be able to specify that their script is a *bash* script, and run *PaSh* without it erroring. Moreover, similar to the POSIX-compliant *PaSh*, the *bash*-compliant *PaSh* should be correct and prove to achieve similar speedups in most cases.

In the following chapter, Chapter 2, we provide relevant context and history that led up to and motivated the *Bash*-compliant *PaSh*. In Chapter 3, we discuss, in-depth, the design of the code infrastructure that supports this work and the

challenges we faced while developing this infrastructure. In Chapter 4 we describe how we evaluate both the correctness and efficiency of the *Bash*-compliant *PaSh*. In Chapter 5 we discuss limitations in this work, including known bugs in other systems used by the *Bash*-compliant *PaSh* as well as limitations inherent to the program. In Chapter 6 we discuss future work that can be done to further develop this area of research, and finally we conclude in Chapter 7.

Chapter 2

Background

In this Chapter, we discuss the most important context and motivation for the *Bash*-compliant *PaSh*. To begin, we consider the historical development of the *shell* and the POSIX standard. Next, we discuss the *bash* interpreter and its importance today. Finally, we describe *PaSh*, a tool for parallelizing POSIX *shell* scripts.

2.1 Shell and POSIX

The first *shell* implementation is attributed to Ken Thompson in 1971, which he created for the Unix operating system [9]. Its purpose was to allow users to interact with the operating system in a concise yet powerful way. Some of the earliest features, which still exist in modern *shell* interpreters today include [10, 11]:

```
echo hi > file.txt
```

The output redirection (`>`) tells the *shell* interpreter to redirect the output that it would print to the screen (write to the output file descriptor) and instead open `file.txt` and write to it. In this case, `echo hi` tells the terminal to write `hi` to the standard output by default, but instead `hi` is written to `file.txt`.

```
cat file.txt | grep word
```

The pipeline operator (`|`) tells the *shell* interpreter to take the output of the first command and treat it as the input for the second command. So in this case,

`cat file.txt` reads the contents of `file.txt` and then `grep word` searches for `word` in the read contents of `file.txt`. Pipelines in *shell* scripting are a powerful tool for data processing.

```
some-long-command &
```

The trailing ampersand (`&`) at the end of a command tells the *shell* interpreter to run the command in the background. Normally, when a command in the *shell* interpreter is running the *shell* waits for it to finish before executing the next, however, when an ampersand is added this is no longer the case. This is a powerful utility when a command might take a while to run and the user wants to run other commands concurrently.

Over the next decades as Unix operating systems and *shell* interpreters continued to develop, the POSIX standard was introduced in 1988 to define a core set of features that these operating systems should have. In 1992, this standard was extended to include a framework for *shell* interpreters. The most recent update to this standard was in 2017. Features that all *shell* interpreters should have according to this standard include [12]:

```
if some-command ; then
    some-other-command
else
    some-other-command-2
fi
```

The if statement in the *shell* programming language allows users to execute different commands depending on the truthiness of a condition.

```
for j in $(seq 0 24); do
    echo hi
done
```

Similar to other languages, *shell* interpreters should have a for loop. This code will loop from numbers 0 (inclusive) to 24 (exclusive) and print `hi` each time.

```
var=hi
echo $var
```

```
echo $(cat file.txt)
```

Another powerful feature of the POSIX *shell* specification is the `$`, which has several special meanings. In the case of `echo $var`, the dollar sign tells the *shell* interpreter to treat `var` as a variable name rather than a string literal, meaning that `hi` will be printed since it was defined as the value of `var` above. On the line below `$(cat file.txt)` means that the string inside of `$()` should be treated as a command to be executed and the output should be expanded as the value. So in this case the contents of `file.txt` will be printed on the screen.

The POSIX specification defines many more *shell* programming language features that encapsulate the most commonly used features in today's modern *shell* variants. In the next section, this dissertation will describe *bash*, the most popular modern *shell* interpreter.

2.2 Bash

In 1977, six years after the release of the first Unix *shell* interpreter, Stephen Bourne developed the *Bourne Shell*. This *shell* interpreter was intended to be an upgrade from the previous, introducing features like loops and variables, which did not exist in the earliest *shell* interpreters. Over the next decade, progress on *shell* variants continued and in 1989 Brian Fox released the *Bourne-Again Shell (bash)* [2].

Although there are other *shell* implementations, such as *zsh*, the default on MacOS, and *PowerShell*, the default on Windows, *bash* is the default on most major Linux distributions [13]. Beyond the POSIX-specified features, *bash* has other useful features including [14, 15]:

```
if (( 2 + 2 )) ; then
    echo hi
fi
```

The arithmetic operator `(())` tells *bash* to treat `2 + 2` as a mathematical expression to be evaluated as either true or false. In this case, 4 is truthy so `hi` will be printed to the terminal.

```
my_array=(str1 str2 str3)
```

In *bash*, users can define arrays using `()`.

```
[[ 1 = 1 ]]
```

Bash also has powerful conditional expressions that allow users to evaluate mathematical expressions as well as conditions such as whether something is a file or a directory.

Beyond this, *bash* has many more non-POSIX features however discussing them all would be beyond the scope of this paper.

2.3 PaSh and Parallelization

Today, the *shell* continues to be one of the most popular programming languages in the world. However, computer scientists and software engineers aren't the only users of the *shell* programming language. In fact, the *shell* is quite popular in many fields that require heavy data processing such as computational biology and physics. There are several reasons for its prevalence in these fields. For one, the *shell* is portable. A user can take their *bash* script on one computer and go run it on another computer without worrying about downloading a new compiler or interpreter—essentially every operating system comes with a *shell* interpreter (usually *bash* on Linux). Moreover, it is powerful for data analysis. Consider a command line utility such as `awk`. `awk` is a utility that is used in the *shell* programming language for finding data in files and processing it in a variety of ways. In one line, `awk` followed by a few more words can do a wide array of tasks that might take 50 lines to write in another programming language. The downside of the *shell* programming language is that there can be a steep learning curve to become very well-versed, however, for someone who works extensively in data processing, this is often worth it.

Writing data processing scripts becomes much more complicated, however, when the programmer wants to speed up their script with parallelization. Attempting to use multiprocessing in a Bash script is, for one, incredibly error-prone. Moreover, attempting to add parallelization can make an elegant script

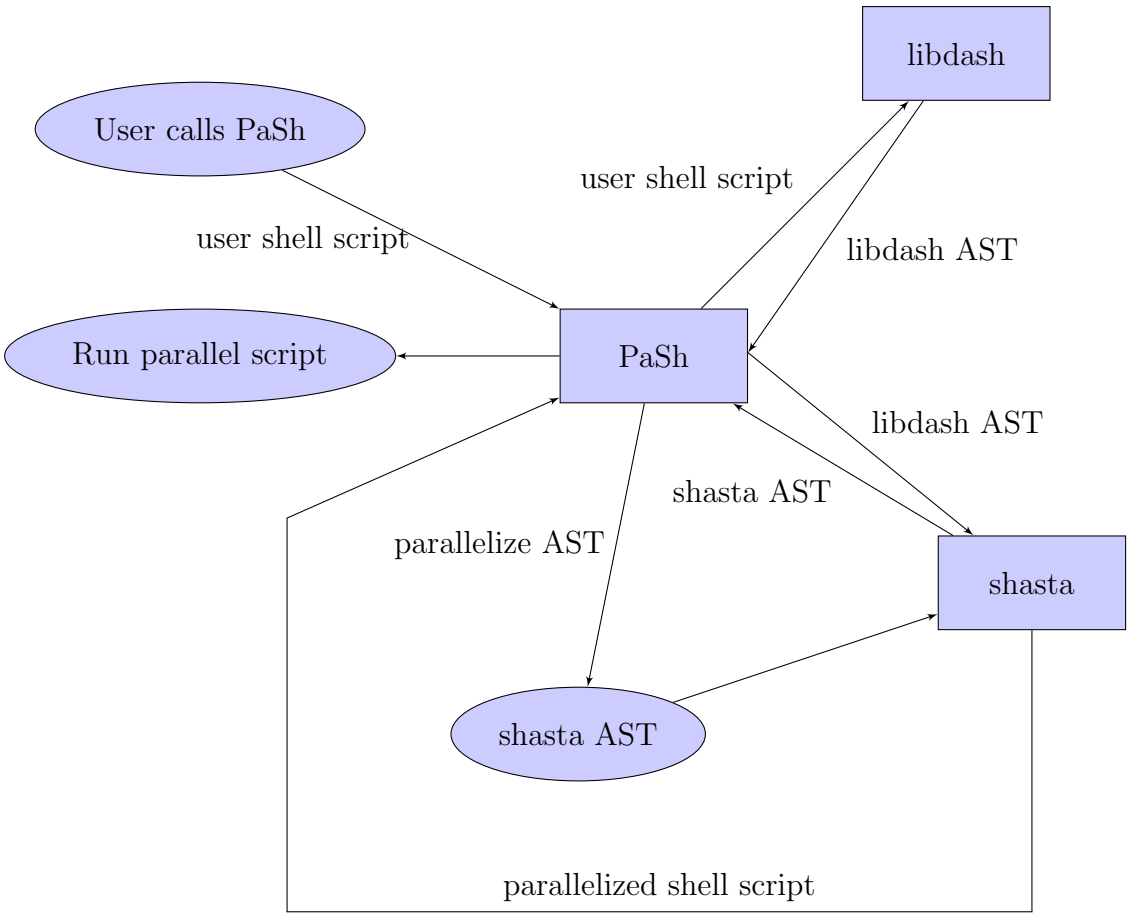


Figure 2.1 High-level PaSh Diagram

look very messy, making it difficult to decipher what the code is doing. For users focused on the core functionality of their script, adding parallelization is risky, time-consuming, and impractical.

In 2021, Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković, created *PaSh* to alleviate this issue. *PaSh* converts a POSIX *shell* script to a dataflow graph, performs semantics-preserving transformations on the graph, converts that graph back to a *PaSh* script, and runs it [6]. Consider the following example:

```
cat f1 f2 | grep "foo" > f3 && sort f3
```

This script first reads the contents of files `f1` and `f2`, then it passes those contents via a pipe to `grep "foo"` which searches for lines containing the word `foo` in these contents. Then it writes the results to file `f3` via output redirection. Finally, the file `f3` is sorted alphabetically. An optimization that *PaSh* might

perform on such a script would be to instead run `grep "foo"` on each of `f1` and `f2` in parallel, and then `cat` the result, redirect it to file `f3` and then `sort f3`.

Though these transformations are the core of what makes *PaSh* such a powerful tool, several other major components make up *PaSh*, highlighted in Figure 2.1. To start, a user calls *PaSh*, passing as input their POSIX shell script. Then, *PaSh* passes off the script to *libdash*, a tool that converts a POSIX script into an Abstract Syntax Tree (AST) representation in *python* [16]. Next, *PaSh* takes this AST and passes it to *shasta* to convert it into a different AST representation [17]. While *libdash*'s AST is more general purpose, the *shasta* representation is tailored to *PaSh's* needs. Once the *shasta* AST is obtained, it is analyzed by *PaSh* for parallelizable components and then transformed. The transformed AST is passed back to *shasta* which prints the AST back to a *shell* script. Finally, this *shell* script, with parellization built in, is run.

In later sections, this dissertation will discuss how these components work in more detail as they relate to the Bash-compliancy integration.

Chapter 3

Design

The goal of this research is relatively simple — take *PaSh*, a tool that parallelizes POSIX *shell* scripts, and extend its functionality to support *bash* scripts while maintaining correctness and speed-up benefits. However, as this dissertation showed in the previous section, many components make up *PaSh*, most of which assume that they are dealing with POSIX scripts. To complete the task at hand, these components needed to be either modified or replaced to support *bash*.

One of the first execution steps in *PaSh* is to convert the input shell script to an Abstract Syntax Tree (AST). As discussed in Section 2.3, in the current version of *PaSh*, this is done by *libdash*, a Python library for parsing *dash* shell scripts. The *dash* shell is a POSIX-compliant but minimal shell interpreter — that is, it has very few additional features beyond those defined in POSIX. This library takes an interesting approach to parsing. Rather than doing parsing directly, *libdash* invokes the *dash shell* interpreter's internal parser. Since parsing is a complex and error-prone task, it is easier and safer to use a complete and well-tested parser.

This parser, however, is only for *dash* scripts. As a first step in completing this thesis, we created *libbash*, a similar tool but for *bash* rather than *dash*. Since *bash* has its own interpreter and also uses a somewhat different AST representation, we published *libbash* as an independent tool.

Once *PaSh* obtains the initial AST, it converts it to a *shasta* AST, an intermediate AST used by *PaSh*. While this framework closely matches the AST generated by *libdash*, it is different than the *libbash* AST. Therefore, the second major component of this work was integrating *libbash* into *shasta*, which involved

two main sub-components. The first sub-component was a function that converts a *libbash* AST to a *shasta* AST. The second sub-component was updating the *shasta* AST to support the additional features that *bash* has.

Once *PaSh* obtains the *shasta* AST, it uses it to do transformations. For the final component, we integrated these changes into *PaSh*— this was a smaller component than the other two because no new work on transformations was needed. The goal of *Bash*-Compliant *PaSh* was not to add new parallelization techniques for *bash* scripts, but rather to run *PaSh* on *bash* scripts without breaking, while using the existing parallelization techniques.

When *PaSh* obtains the *shasta* AST, it first does a preprocessing step where it converts the AST to an Intermediate Representation (IR). This IR is very similar to the AST, except that it contains holes—areas that we would like to identify as potential parallelization areas. Since we have updated *shasta* with new nodes we need to add support for these nodes in this preprocessing step.

When *PaSh* actually runs the script, it uses these holes as points in the script to call *PaSh*'s Just-In-Time engine—code called during the script execution where parallelization is done. This is discussed in much greater detail in Section 3.3.2, but some small changes are needed here for correctness on *bash* scripts.

3.1 libbash

The high-level goal of *libbash* is simple — a user should be able to give it a Bash script and get an Abstract Syntax Tree (AST) representing that script in return. Since Bash parsing is a long and difficult task, we use the *bash* interpreter's internal parser to obtain the AST. Because this library is in *python*, there are two major components that make up this work. The first is configuring the *bash* source code to allow us to directly obtain this AST without actually running the *bash* script. The second component is getting this AST from *C*, the programming language that *bash* is written in into *python*.

We published *libbash* an independent *python* library hosted on PyPI [18, 19].

3.1.1 Bash Source Code

The *bash* interpreter is a code base written in *C* that compiles to an executable [20]. When a user downloads the *bash* source code, they must first run an executable called `configure`. This script checks a list of macros (which the user may change) and creates additional *C* programs, a `Makefile`, and a few other files. The `Makefile` is created based upon these macros as well as a template called `Makefile.in`. Once the `Makefile` has been generated the user may run the `make` command which will generate the *bash* executable. This executable is the program that is *bash*.

The first change we made is with `Makefile.in`. Since we wanted to be able to invoke *bash*'s parser directly, we must make *bash* a shared object. A shared object is not directly executable but instead contains functions that may be invoked from other programs. Since the `Makefile` uses `gcc` to compile *bash*, we simply passed the `-shared` flag into this command so that `gcc` produces a *bash* shared object instead. We also passed the `-fPIC` flag which causes position-independent code to be generated. This is needed for compatibility with different operating systems.

The *bash* interpreter has a `main` function which runs the program. This function is implicitly called when a user opens a new terminal on their computer, and it continues to run until the user exits their terminal. Within this function, a large amount of environment setup is done. Moreover, arguments to the program are parsed and used. While much of this setup is only relevant for script execution, some is needed for parsing. Therefore, we needed to create a function that does just the necessary setup for parsing. Below is the that function:

```
// libbash - does the neccesary initialization for the shell
// normally done in main()
int initialize_shell_libbash(void)
{
    int code = setjmp_nosigs(top_level);
    if (code)
        return (EXECUTION_FAILURE);

    set_default_locale();
```

```
    // should we consider the user's environment?
    if (getenv("POSIXLY_CORRECT") || getenv("POSIX_PEDANTIC"))
        posixly_correct = 1;

    set_shell_name("bash"); // turns out this is important - we
       ↪ replace argv[0] with bash, hopefully this works

    init_noninteractive(); // don't think we need to worry about
       ↪ init_interactive

    /* If we're in a strict Posix.2 mode, turn on interactive
       ↪ comments,
       alias expansion in non-interactive shells, and other Posix.2
          ↪ things. */
    if (posixly_correct)
    {
        bind_variable("POSIXLY_CORRECT", "y", 0);
        sv_strict_posix("POSIXLY_CORRECT");
    }

    int should_be_restricted;
#if defined(RESTRICTED_SHELL)
    should_be_restricted = shell_is_restricted(shell_name);
#endif
#if defined(RESTRICTED_SHELL)
    initialize_shell_options(privileged_mode || restricted ||
       ↪ should_be_restricted || running_setuid);
  initialize_bashopts(privileged_mode || restricted ||
     ↪ should_be_restricted || running_setuid);
#else
    initialize_shell_options(privileged_mode || running_setuid);
```

```
    initialize_bashopts(privileged_mode || running_setuid);
#endif

    if (shell_initialized) {
      shell_reinitialize();
    } else {
      shell_initialize();
      shell_initialized = 1;
    }

    set_default_lang();

    set_default_locale_vars();

    /* If we are invoked as ‘sh’, turn on Posix mode. */
    if (act_like_sh)
    {
        bind_variable("POSIXLY_CORRECT", "y", 0);
        sv_strict_posix("POSIXLY_CORRECT");
    }

    cmd_init(); /* initialize the command object caches */

    uwp_init();

    return 0;
}
```

Some of the setup done here includes setting the *shell* name and setting local variables. The process for writing this function was more so trial and error than anything else. We started with much more than we needed and iteratively removed lines of code while ensuring we could still invoke the parsing function, which we describe later.

Besides doing the initial *bash* setup, we also needed to set the file storing the script and do a bit of parser setup. Below is the function that does this:

```
// libbash - tells the shell which file to read the commands from
// returns zero on success, negative on failure
int set_bash_file(char *filename)
{
    // it seems like when multiple calls to bash_to_ast are made in
       ↪ the same process
    // the line number is not reset, so we do it here
    line_number = 0;
    EOF_Reached = 0; // same for EOF_Reached

    reset_parser();

    shell_script_filename = filename;
    int open_sh = open_shell_script(filename);
    if (open_sh < 0)
        return open_sh;

    set_bash_input(); // seems neccesary based on comments
    return 0;
}
```

Here we set the `shell_script_filename` variable used globally. We also need to make sure to reset `line_number` and `EOF_Reached` because if multiple calls to the parser are called in the same process (which would happen if the parsing function is called several times in a Python program) the same Bash shared object instance is used. Since these variables are used in parsing, they need to be reset.

In *bash*, once setup is complete, a function titled `reader_loop` is called. This function loops over user input (or reads a line from a *bash* script), parses it, and executes it. The function that reads the line from the script and parses it is titled `read_command`. We wrote a small wrapper function around `read_command` to use in *libbash*. To parse, *bash* invokes a *Yet Another Compiler-Compiler* (YACC)

program to parse the script. YACC is a tool that allows the programmer to specify the syntax of a language and parse programs in that language into an Abstract Syntax Tree [21]. When a line of the Bash script has been parsed the global variable `global_command` is set to the AST. Once the entire script has been parsed the next call to `read_command` sets `global_command` to `null` which allows *libbash* to recognize that the entire script has been parsed.

The internal AST structure in *bash* is defined by the `command` struct in the source code:

```
/* What a command looks like. */
typedef struct command {
  enum command_type type; /* FOR CASE WHILE IF CONNECTION or SIMPLE
      ↪ . */
  int flags;   /* Flags controlling execution environment. */
  int line;   /* line number the command starts on */
  REDIRECT *redirects;  /* Special redirects for FOR CASE, etc. */
  union {
    struct for_com *For;
    struct case_com *Case;
    struct while_com *While;
    struct if_com *If;
    struct connection *Connection;
    struct simple_com *Simple;
    struct function_def *Function_def;
    struct group_com *Group;
#if defined (SELECT_COMMAND)
    struct select_com *Select;
#endif
#if defined (DPAREN_ARITHMETIC)
    struct arith_com *Arith;
#endif
#if defined (COND_COMMAND)
    struct cond_com *Cond;
```

```
#endif
#if defined (ARITH_FOR_COMMAND)
    struct arith_for_com *ArithFor;
#endif
    struct subshell_com *Subshell;
    struct coproc_com *Coproc;
  } value;
} COMMAND;
```

The `command_type` specifies what type of command it is, such as a `for` loop or a `simple` command. The `flags` field specifies whether certain special rules should be applied to this command. For example, the `CMD_INVERT_RETURN` flag means that the exit code resulting from executing this command should be inverted. The `CMD_TIME_PIPELINE` flag means that the amount of time it takes to execute this command should be recorded. The `line` field is unused (line numbers are stored in specific command structs). The `redirects` pointer stores a list of redirections in the command. However, some specific command types store their own redirects. The `union` of different command types stores the specific type of command. The `command_type` enum specifies which of the values in the union is set.

The AST has a recursive structure where `simple_com`, and a few others, act as the base commands and other commands recursively have `command`s as fields. For example, while loops in *bash* are defined by the following struct:

```
/* WHILE command. */
typedef struct while_com {
  int flags;   /* See description of CMD flags. */
  COMMAND *test;  /* Thing to test. */
  COMMAND *action;  /* Thing to do while test is non-zero. */
} WHILE_COM;
```

Similar to the top-level command, the `while_com` stores flags for special cases. It also stores two pointers to the top-level `command` struct. The `test` is the command to execute every time we loop to determine if we should continue looping and `action` is the thing to execute in the loop. So for example, in the following

script `[ $x -le 5 ]` is the `test` and `echo "Welcome $x times"` is the `action`.

```
x=1
while [ $x -le 5 ]
do
  echo "Welcome␣$x␣times"
done
```

The `simple` command has the following structure:

```
/* The "simple" command. Just a collection of words and redirects.
   ↪ */
typedef struct simple_com {
  int flags;   /* See description of CMD flags. */
  int line;   /* line number the command starts on */
  WORD_LIST *words;  /* The program name, the arguments,
                                variable assignments, etc. */
  REDIRECT *redirects;  /* Redirections to perform. */
} SIMPLE_COM;
```

The `flags` stores command flags and `line` is the line number the command is on. The field `words` is essentially the list of space-separated words in the command, however, there are nuances to this rule. For example, for the command `echo "Welcome $x times"` the word list would contain `echo` as the first word and `"Welcome $x times"` as the second since quotes tell *bash* to treat everything inside them as one argument. The `redircts` field stores the list of redirections for this simple command specifically.

The `word` in *bash* is represented by the following struct:

```
/* A structure which represents a word. */
typedef struct word_desc {
  char *word;  /* Zero terminated string. */
  int flags;  /* Flags associated with this word. */
} WORD_DESC;
```

It contains the pointer to characters that contain the `word` and it also contains

flags for this word. Examples of these flags include `W_HASDOLLAR` which specifies that a dollar sign is present in the word and `W_QUOTED` which specifies that the word is quoted. These flags act as hints rather than explicit instructions for how to execute the command—the `W_HASDOLLAR` flag, for example, tells the *bash* program execution function that the word may need to be expanded, but not that it must be expanded.

3.1.2 Python interface

Once we exposed the functions needed to parse a *bash* script, the next task was to write a function to get the AST for a *bash* script in *python*. To do this, we used *ctypes*, a *python* library that allows the user to load shared objects, call functions, and access variables in them [22]. This allowed us to write a function called `bash_to_ast`, which takes as input the name of a file containing a *bash* script and returns a list of `Command` objects. This function calls the *bash* setup functions, and then repeatedly calls `read_command`, and extracts the result set to `global_command` until *bash* has finished reading through the script file.

The *python* interface also needed to have an AST that matched the AST structure in the *bash* source code. To do this, *libbash* uses a recursive class structure, which as much as possible mirrors the recursive struct structure in the bash interpreter. For example, the `Command` class in *libbash* is:

```
class Command:
    """
    a mirror of the bash command struct defined here:
    https://git.savannah.gnu.org/cgit/bash.git/tree/command.h
    """
    type: CommandType # command type
    flags: list[CommandFlag] # command flags
    # line: int # line number the command is on - seems to be
        ↪ unused
    redirects: list[Redirect]
    value: ValueUnion
```

The main difference in the Python representation is the use of more modern data structures. For example, `flags` is a list of objects rather than an integer where each bit represents whether a flag is set.

Converting `command` structs in `ctypes` to the `Command` class is done via the `Command` constructor, which recursively calls on its field's constructors to build the `Command`. As an example, here is the constructor for the `while` command Class:

```
def __init__(self, while_c: c_bash.while_com):
    """
    :param while_c: the while command struct
    """
    self.flags = command_flag_list_from_int(while_c.flags)
    self.test = Command(while_c.test.contents)
    self.action = Command(while_c.action.contents)
```

Here, there is a one-to-one mapping of fields between the *C* and *python* representation. However, the `flags` field is created by a function that converts the `flags int` into a list of `Command Flags`. The `test` and `action` are passed recursively to the `Command` constructor.

Besides `bash_to_ast`, there are a few more functions we added to *libbash*, though `bash_to_ast` is the only function used in *PaSh*. The first is `ast_to_bash`, a function that takes the AST representation of a *bash* script and converts it back to a *bash* script. This function is not guaranteed to produce the exact same script that it was generated from, because stylistic elements are not stored in the AST. However, it should produce a semantically equivalent script—that is, the script that generated the AST and the script generated from the AST should have the exact same effect and result when run. This function, similar to `bash_to_ast`, actually invokes the *bash* source. In *bash* there is a function called `make_command_string` which takes a `command` struct and returns a `string`. To invoke this function, the `Command` object in *python* must be converted back to `ctypes` types and then passed into this function.

We also wrote a function called `ast_to_json`, which takes a list of AST objects and converts them to an equivalent representation, except with primitive types —

lists, dictionaries, integers, strings, and None. This function primarily serves to allow a `Command` list to be printed and interpreted manually by a user.

Finally, we wrote an equality operator on `Commands`. This operator recursively checks if two `Command` objects are equivalent, ignoring stylistic differences. This tool can help determine if two *bash* scripts are semantically equivalent (though there are cases where functionally equivalent scripts may appear different, the inverse case should never happen).

3.2 Shasta Integration

Shasta is a *python* library providing an AST representation for *shell* scripts [17]. Though it was originally part of `PaSh` it was separated into its own library so it could be used for other tools. Because it was created for *PaSh*, the *shasta* AST closely matches the *dash* AST used in *libdash*. For *bash* scripts to be supported in *PaSh*, we needed to add additional features of the *bash* AST to the *shasta* AST while maintaining backward compatibility for *PaSh* in POSIX mode.

To complete this integration, there were two major sub-components. The first was writing a function that converts a *libbash* AST into a *shasta* AST. The second sub-component was changing the *shasta* AST to support *bash*'s non-POSIX features.

3.2.1 Libbash to Shasta

To convert a *libbash* AST to a *shasta* AST, we wrote a function called `to_ast_node`. This function recursively traverses a `Command` to generate the AST. While the two ASTs share a largely similar recursive structure, some differences require careful consideration to maintain AST correctness.

One of the strategies we employed in preserving correctness was looking at *bash's* `make_command_string` function to determine which fields are actually used for printing a script from the AST. For example, consider our helper function for converting a *libbash* `while` node into a *shasta* while node:

```
def to_while_node(node: WhileCom) -> WhileNode:
    test = node.test
```

```
    body = node.action
    return WhileNode(
        test=to_ast_node(test),
        body=to_ast_node(body))
```

Although the *libbash* `WhileCom` also contains a `flags` field as shown in code block 3.1.1, it is completely unused in the printing function — it is just used for hinting in *bash*'s execution function.

Another distinction is that *shasta* does not have a top-level `Command` struct, but instead just has different `command` types. In *libbash*, in cases where the top-level `Command` has redirections, the top-level `command` in *shasta* is a `RedirectionNode`, which just stores a list of redirections and a `command`.

Another distinction is that the *shasta* AST stores assignments in simple commands. Thus, we wrote a function called `to_assign_node` which takes a `WordDesc` and converts it to an assignment, if one exists.

Shasta also represents words as lists of `ArgChars`. An `ArgChar` is usually just a character, however, they are also used to represent expansions. For example, the word `$var` would be represented by a single `VArg`, which represents words to be variable expanded. The `CArg` is used to represent simple characters. The function that converts *libbash* ASTs to *shasta* ASTs represents every character as simply a `CArg`. This is not technically correct, because *PaSh* relies on these `ArgChar` types to do its analysis. We employ a workaround in *PaSh* which we discuss later in Section 3.3.2. This is also discussed in greater detail as an area of future work in Chapter 6.

3.2.2 Adding Shasta Nodes

Since *bash* is a superset of the POSIX standard, we needed to add some node types to *shasta*, and we needed to modify others to support these new features. In some cases, this addition was quite simple. POSIX, for example, does not have `select` commands, which allow a user to make menus in the terminal. They also don't have `arithmetic` commands, which allow users to evaluate mathematical operations in conditional statements. Thus, we added several new nodes to *shasta*.

This is the `SelectNode`:

```
class SelectNode(Command):
    NodeName = 'Select'
    line_number: int
    variable: list[ArgChar]
    body: Command
    map_list: list[list[ArgChar]]
```

It mirrors the `SelectCom` in *libbash* except with *shasta* types.

While creating new nodes was relatively simple, deciding how to modify existing nodes was a more complicated task. Perhaps the most interesting of these integrations was with redirections. In the original version of *shasta* there exist three types of `RedirNodes`. The first is the `FileRedirNode`:

```
class FileRedirNode(RedirectionNode):
    NodeName = "File"
    redir_type: str
    fd: int
    arg: "list[ArgChar]"
```

The `redir_type` field specifies the type of file redirection, such as input redirection and output redirection. The `fd` is the file descriptor to be redirected. In practice, this number is often omitted in the script. For example, the redirection `> file` says to redirect output from the script to `file`, and here the default output file descriptor is 1. So this redirection is semantically equivalent to `1> file`. Finally, the `arg` is the file to redirect to.

While the file redirections in *bash* are mostly POSIX, *bash* supports an additional feature for all redirections where a user can specify a variable name that will be assigned to a file descriptor greater than 10 [10]. This variable name will remain assigned to the file descriptor for the remainder of the program. To handle this option, we changed `fd` to a `(str, [list[ArgChar], int])`. The first part of this tuple specifies if the file descriptor is a number or if it is a variable name and the second value is either that number or variable name.

The second type of redirect in *shasta* is the `DupRedirNode`:

```
class DupRedirNode(RedirectionNode):
    NodeName = "Dup"
    dup_type: str
    fd: int
    arg: "list[ArgChar]"
```

Duplication redirections, in general, allow the user to duplicate a file descriptor. In POSIX, there are only two types of duplication redirections. The first, `[n]>&fd` says that we want to duplicate `fd` to the same output file descriptor as n, or standard output if unspecified. The second, `[n]<&fd` says that we want to duplicate `fd` to the same input file descriptor as n, or standard input if unspecified. In *bash* there are many more duplication redirection types than defined by POSIX. In the updated version of *shasta*, we define the `DupRedirNode` as following:

```
class DupRedirNode(RedirectionNode):
    NodeName = "Dup"
    dup_type: str
    fd: (str, [list[ArgChar], int]) # either ('var', variable_name)
        ↪  or ('fixed', fd)
    arg: (str, [list[ArgChar], int]) # either ('var', variable_name
        ↪ ) or ('fixed', fd)
    move: bool
```

While the `dup_type` options are the same two, the other field changes specify different types of redirections. Firstly, `fd` and `arg` can also be a file name because of *bash*'s support for creating variables that refer to file descriptors. Additionally, the `move` boolean specifies if a `-` should be put at the end of the redirection, which changes its behavior. When this is done, the file descriptor is closed after being duplicated.

The third redirection in *shasta* is the `HeredocRedirNode`:

```
class HeredocRedirNode(RedirectionNode):
    NodeName = "Heredoc"
    heredoc_type: str
```

```
    fd: int
    arg: "list[ArgChar]"
```

The `heredoc_type` field specifies some details about quoting but this `Node` essentially specifies one type of POSIX redirection, heredocs. A heredoc takes the following form:

```
cat << EOF
Hello
World
EOF
```

Here, the `<<` tells the *shell* that the Heredoc is starting. The word `EOF` specifies the start of string literals in the script. The second `EOF` specifies the end of the string literals in the script. The data in between is treated as contents of a file and passed as an argument to `cat`. So the result of running this script is that `Hello` and `World` are written to output. Bash, however, supports additional types of Heredocs. The updated `HeredocRedirNode` class is:

```
class HeredocRedirNode(RedirectionNode):
    NodeName = "Heredoc"
    heredoc_type: str
    fd: (str, [list[ArgChar], int]) # either ('var', filename) or
        ↪ ('fixed', fd)
    arg: "list[ArgChar]"
    kill_leading: bool
    eof: [str, None]
```

Again, `fd` must also support variable names. Additionally, there is a `kill_leading` boolean which indicates that tabs in the string literal should be ignored. This allows a user to add tabs before there Heredoc string literal for style without literally including those tab characters as part of the string. The `eof` field is a stylistic addition to `Bash` that we chose to keep.

Though these updates to redirections in *shasta* cover most of the additional redirections Bash provides, a few miscellaneous redirections that weren't covered

by the previous nodes were added as the `SingleArgRedirNode`:

```
class SingleArgRedirNode(RedirectionNode):
    NodeName = "SingleArg"
    redir_type: str
    fd: (str, [list[ArgChar], int]) # Either ('var', filename) or
        ↪ ('fixed', fd)
```

This node represents three redirection types. The first, `fd>&-` just asks Bash to close `fd`. The second, `&>file` specifies that we'd like to redirect standard output and standard error file descriptors to `file`. The third, `&>>file` is similar to the second, except that we'd like to append data written rather than potentially overwriting the contents of `file`.

There were also challenges with integrating other types of nodes. For example, in *bash*, the ampersand can be used as a separator between commands, whereas in *dash* the `BackgroundNode` doesn't contain more than one `Command`. This required adding an additional field to the `BackgroundNode`, and it required printing in different ways depending on if this extra `Command` is set.

3.2.3 Printing Shasta Nodes

While a primary motivation for *shasta* is to act as an AST for *PaSh* to use, once the parallelization transformations have been applied to the AST, *shasta* is responsible for printing the AST back to a script. The challenge of integrating script printing that is correct for both *dash* and *bash* was certainly the most challenging part of the *shasta* integration and required some clever tricks to implement.

One of the trickiest things to implement was printing with `SemiNodes`. The `SemiNode` represents commands separated by semicolons, and in some cases newlines. In the old version of *shasta*, these are always printed with braces surrounding, for example:

```
{echo hi ; echo bye}
```

However, this syntax is not always valid in *bash*. For example, the following command tells *bash* to run `echo hi` and then `echo bye` and do it in a subshell.

```
( echo hi ; echo bye )
```

While the above syntax is valid, this syntax is not:

```
( { echo hi ; echo bye } )
```

There are several other node types for which printing child `SemiNode`s with braces is invalid. However, never printing them with braces leads to issues with compatibility with *libdash*. Thus, for several node types, we recursively checked if its child was a `SemiNode`, and if it was made sure to not print the braces.

There were a few similar cases of situations where nodes should be printed differently depending on their parent that we also needed to consider.

Another major challenge was printing heredoc redirections. Heredocs are particularly tricky when used in pipelines. In a pipeline, when the first command was a simple command, we needed to extract all the heredoc redirections from it, if any. We needed to print the headers before the pipeline started and print the body of the heredocs after the pipeline. For example, consider the following command:

```
cat << EOF | grep word
word
EOF
```

This command outputs `word` and then `grep` searches for the string `word` in the text and finds `word`. However, in isolation, the redirection would be printed as:

```
<< EOF
word
EOF
```

By default, commands just print redirections after all of the words as this is usually correct. However, this is actually invalid syntax:

```
cat << EOF
word
EOF | grep word
```

Thus, we needed to separate the printing of `HeredocRedirNodes` into the printing of the header and printing of the body, and in the case of `PipeNodes` print the headers after the first command and the body last. This was also needed for `AndNodes`— commands separated by `&&`.

Besides these challenges, the printing infrastructure needed to be updated to support new node types as well as new features of existing nodes.

3.3 PaSh Integration

The final component of the *Bash*-compliant *PaSh* project was integrating the work on *libbash* and *shasta* into *PaSh*. The *PaSh* user can pass in a `--bash` flag and *PaSh* should process the script as a *bash* script.

This component of the project was the simplest because we did not add any new parallelization techniques to *PaSh*. So although *shasta* has new node types, *PaSh* can ignore them. *PaSh*'s parallelization techniques are only on simple commands and pipeline commands. So, for example in the following script:

```
for j in $(seq 0 24); do
    cat 1GB.txt | grep $j >> file.txt
done
```

The command `cat 1GB.txt | grep $j >> file.txt` would be identified for potential parallelization, however, the looping logic of the for loop would not.

3.3.1 Preprocessing

The first step for adding *bash* support to *PaSh* was adding support for converting the new node types from *shasta* AST to *PaSh*'s intermediate representation (IR). The IR is equivalent to the AST except for some more interesting stuff happening on simple nodes and pipeline nodes, including marking them as potential areas for parallelization. Thus, to implement preprocessing, we just added functions that recursively descend the new nodes to create the IR. For example, the helper function for creating the IR for a select node calls `preprocess_close_node` which does the more complex work of identifying regions for parallelization.

```
def preprocess_node_select(ast_node, trans_options, last_object=
    ↪ False):
    ast_node: SelectNode = ast_node
    preprocessed_body, sth_replaced = preprocess_close_node(
        ↪ ast_node.body, trans_options, last_object=last_object)
    ast_node.body = preprocessed_body
    preprocessed_ast_node = PreprocessedAST(ast_node,
                                            replace_whole=False,
                                            non_maximal=False,
                                            something_replaced=
                                                ↪ sth_replaced,
                                            last_ast=last_object)
    return preprocessed_ast_node
```

This is all of the work needed to update the preprocessing step of *PaSh*.

3.3.2 Compilation

During runtime, *PaSh* calls a just-in-time *python* program to parallelize. Because we add no new parallelization techniques, very little work is needed on this code. However, as we mentioned in Section 3.2.1, when converting a *libbash* AST to a *shasta* AST we don't identify expansions. Without identifying expansions we cannot correctly analyze whether it is safe to parallelize a script. For example, suppose we have a pipeline with a variable expansion pointing to a file. If we assume that this is plain text, *PaSh* could incorrectly think that it is safe to manipulate the file because it doesn't know that the word actually expands to that file.

To work around this, we pre-expand every word that might be used in parallelization analysis. For each word of a Command that is passed to the Just-in-Time engine, we create a subshell and run the command `echo [word]`. By `echo`-ing the word, we are guaranteed to expand `word`, meaning that it will no longer be expandable. This provides us with the correctness that we need.

Chapter 4

Evaluation

In evaluating the success of this project, we consider two metrics—correctness and efficiency. For correctness, we consider not only the correctness of the *Bash*-compliant *PaSh*, but also the correctness of the *shasta* and *libbash*, the two main tools the *Bash*-compliant *PaSh* uses. For efficiency, there are two main success metrics. Firstly, POSIX-compliant scripts running *PaSh* with and without *bash* mode should run for a similar amount of time. That is, for scripts that the POSIX-compliant *PaSh* obtains speed-ups on, the *Bash*-compliant *PaSh* should also get speedups. Secondly, we should be able to add *bash*-specific components to these scripts and they should still obtain the same speedups on the parallelizable POSIX components.

4.1 Correctness

In the following section, we describe the *libbash*, *shasta*, and *PaSh* testing suites we ran to evaluate correctness.

4.1.1 libbash

To evaluate correctness in *libbash*, we run the following test:

```
try:
    ast = bash_to_ast(test_file)
    ast_to_bash(ast, TMP_FILE)
    bash = read_from_file(TMP_FILE)
```

```
except RuntimeError as e:
    assert str(e) == "Bash␣read␣command␣failed,␣shell␣script␣may␣be
        ↪ ␣invalid"
    continue


ast2 = bash_to_ast(TMP_FILE)
ast_to_bash(ast2, TMP_FILE)
bash2 = read_from_file(TMP_FILE)
assert ast == ast2
assert bash == bash2
```

Here, we begin by converting a *bash* script to its AST. If this operation fails, we assert that the error came from *bash*, meaning that the script was actually invalid. Assuming it wasn't, we then convert it back to *bash*. If both `bash_to_ast` and `ast_to_bash` are correct, the new *bash* script should be semantically equivalent to the original *bash* script. However, it is likely to be stylistically different. Thus, we convert the new *bash* script back to an AST again. Now, we may use the AST equality operator to assert that the old and new AST are semantically equivalent. If this assertion is successful, we can have confidence that `bash_to_ast` and `ast_to_bash` are correct on the script, however, as an additional check we convert the script back to *bash* again and assert that it is equal to the previous iteration of the script. Since both scripts were created by the printing function, these should be stylistically equivalent as well as semantically equivalent.

The *bash* source code conveniently provides a comprehensive testing suite with 473 scripts that we use for testing. Currently, the above test passes on 433 out of 473 tests. Though 40 scripts are currently failing, we have identified that all but 1 are failing because of a few bugs in the *bash* printing function that we invoke. The other failing script is erroring because of a bug in the *bash* parser that we invoke. We have filed bug reports for these errors and note the cases that will not currently work in *libbash*, but will work in the future. This also means that *bash_to_ast*, the only function from *libbash* used in *PaSh* is correct on all but 1 of these scripts, but *ast_to_bash* is incorrect on 39 scripts. These bugs are discussed in detail in Chapter 5.

4.1.2 shasta

The testing process for *shasta* is very similar to the testing process in *libbash*. For converting a script from *bash* to *shasta*, we must first invoke the *libbash* parser, and then we invoke the function that converts the AST to a *shasta* AST. Then, *shasta*'s native printing function is invoked to convert it back to *bash*.

We use the same bash testing suite to test round-tripping scripts in *shasta*. Since we use our own printer, 472 out of 473 scripts pass this test.

4.1.3 PaSh

To test Bash's correctness, we must both ensure that our updates to *shasta* and *PaSh* did not break anything in the POSIX-compliant *PaSh*, and also ensure that *Bash*-compliant *PaSh* works on both *POSIX* and non-*POSIX* scripts. Firstly, we ran the main *PaSh* testing suite not in *bash* mode. The tests run scripts with and without *PaSh* and confirm that, in both cases, the output written to standard output and the exit codes are equivalent. Out of 54 tests, 52 pass. Though 2 are failing, these failures are because of current issues in *PaSh*. We confirmed this by running the same testing suite on an unmodified version of *PaSh* with an unchanged *shasta* and found that the same 2 tests fail on that version as well. The maintainers of *PaSh* also acknowledge that the current version of *PaSh* is expected to have these bugs at the moment.

Since our changes appear not to have changed the functionality of *PaSh* on POSIX scripts, we next ran the same tests with *bash* mode enabled. Just as when running the original version of *PaSh*, 52 out of 54 tests pass, with the same two tests failing.

Finally, to confirm that the *Bash*-compliant *PaSh* is correct on all *bash* scripts, we pulled tests from the *bash* testing suite. However, because of how *PaSh* works we excluded some tests. Firstly, we removed any tests which should fail to parse. When running a *bash* script normally, the script is parsed and executed line-by-line. However, when *PaSh* invokes *libbash* to parse a script, it parses the entire script up front. When running a script with errors without *PaSh* some of the script might execute and write to standard output, whereas with *PaSh* it would

not be run at all. This is the expected behavior of *PaSh*.

The next type of script that must be excluded is any script with commands that change the parsing behavior. For example, consider the following script:

```
shopt -s extglob
ls !(*.txt)
```

The first line tells *bash* that we would like to use extended globbing, meaning *bash* supports additional features for matching names. The second line asks *bash* to list all files in the current directory that do not end in `.txt`. However, without extended globbing `ls !(*.txt)` is invalid syntax. Since *libbash* does not execute commands, it will not know that extended globbing has been enabled and thus, it will throw a parsing error. This is an expected limitation in *PaSh* and is discussed in greater detail in Chapter 5.

Finally, scripts that refer to *bash* variables should also fail. For example, the `BASH_SOURCE` variable is bound to the source file from which the script was run. When running with *PaSh*, we are copying over paralellized scripts to different files, so this variable will differ if we use *PaSh*.

Once we exclude these sorts of scripts everything else should pass. Of the 215 scripts, 203 pass. The remaining 12 scripts fail because they all set the `IFS` variable, an internal variable that tells *bash* how to do word splitting. Setting this variable does not affect parsing, however, setting it does cause *PaSh* to error. This bug, however, has been identified as a bug with *PaSh* rather than any of the *bash* additions, because running the unmodified version of *PaSh* on such scripts produces the same error. A bug report has been filed to have this issue resolved [23].

4.2 Efficiency

To measure the efficiency of the *Bash*-compliant *PaSh*, we use the POSIX-compliant *PaSh* as a baseline. Since we did not add any new parallelization techniques to *PaSh*, we do not expect the *Bash*-compliant *PaSh* to be faster than the POSIX-compliant *PaSh*. However, we do expect *PaSh* to run in a similar amount of time on the same script regardless of whether *bash* mode is enabled. Moreover, we

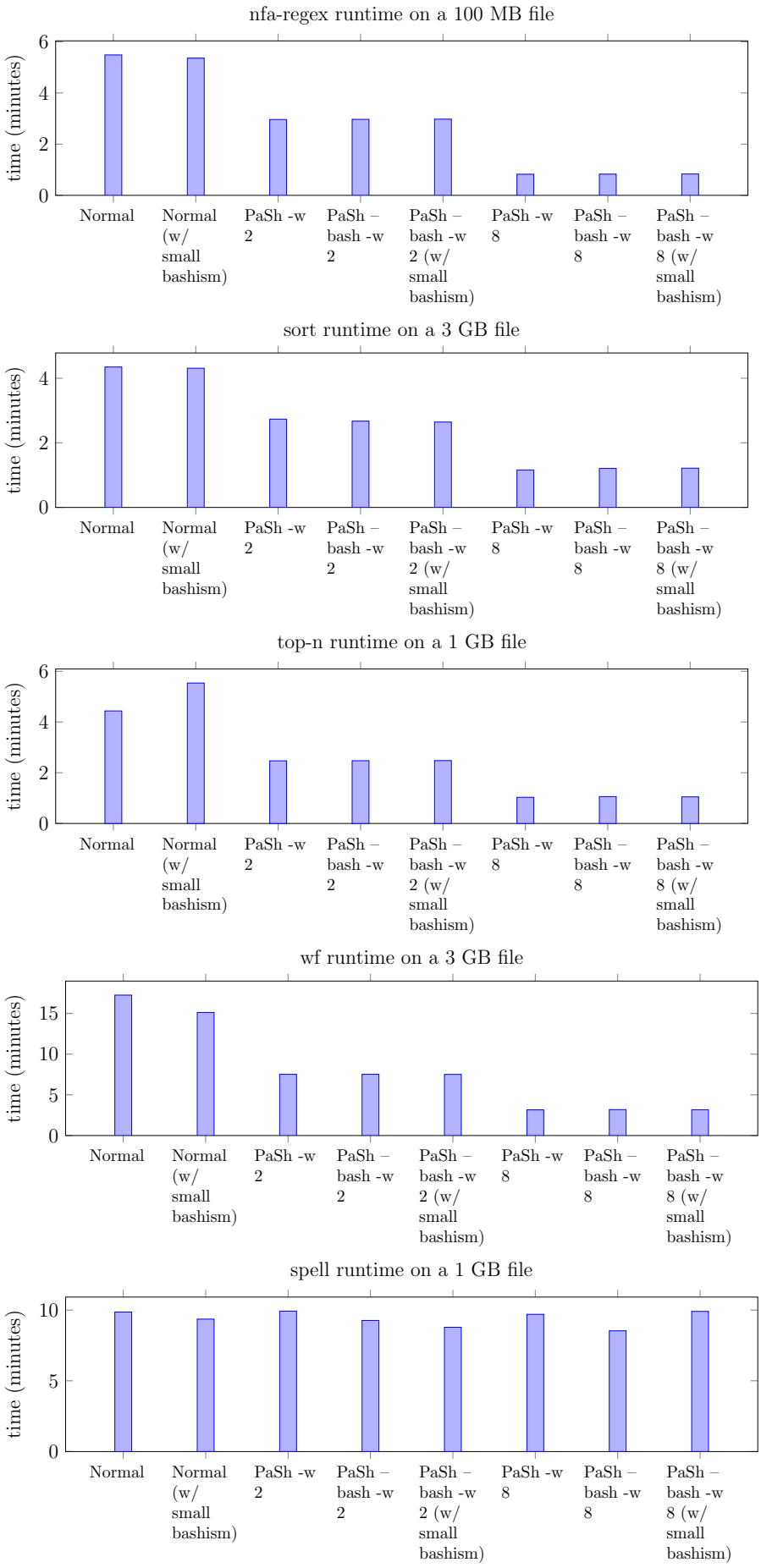


Figure 4.1 Time Benchmark on Oneliner Scripts

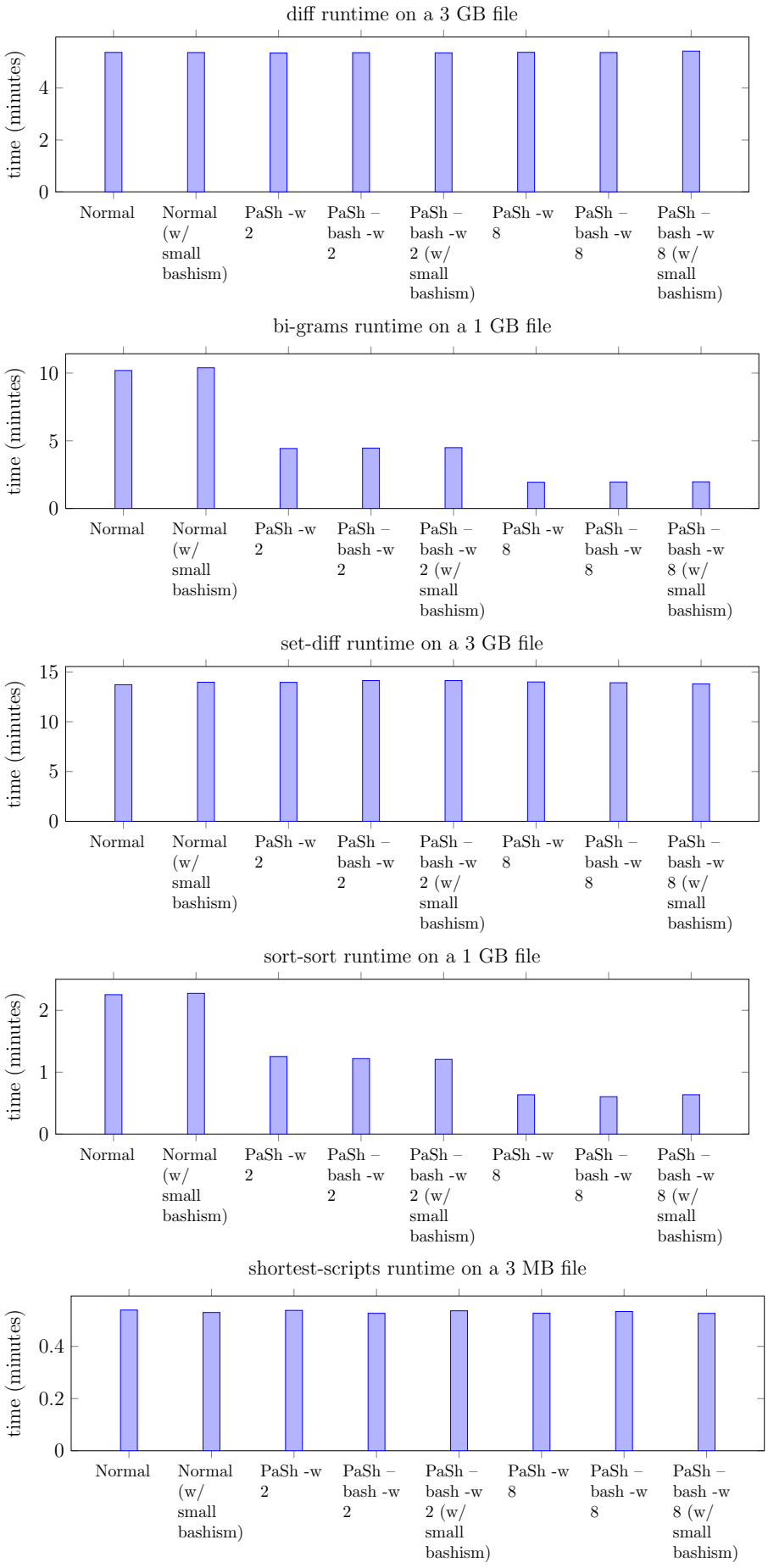


Figure 4.1 Time Benchmark on Oneliner Scripts

should be able to add *bash*-specific features (bashisms) to these scripts and *PaSh* should still get the same speedups on the parallelizable components. To test the former, we simply run *PaSh* on a series of tests with and without *bash* mode enabled and compare the runtimes. To test the latter, we add bashisms that are not computationally expensive (should run in a matter of milliseconds), and compare the runtime to the other results. We also run these tests with different widths, where width specifies how many degrees of parallelization *PaSh* will use.

As benchmarks, we used *PaSh*'s one-liner evaluation suite. This suite consists of 10 scripts that contain one or a few lines of data processing pipelines. When adding bashisms to these tests as part of the evaluation, we used a variety of different things including *bash*-specific command types, redirections, and expansions. The results of these benchmarks are shown in Figure 4.1. These benchmarks show that in all cases, the scripts run without *PaSh* ran in approximately the same amount of time with and without the bashism, as expected. Moreover, with width 2, the POSIX-compliant *PaSh* on the script without the bashism, the *Bash*-compliant *PaSh* on the script without the bashism, and the *Bash*-compliant *PaSh* on the script with the bashism ran in approximately the same amount of time. The same held for width 8. This is precisely the measure of success we hoped for in the *Bash*-compliant *PaSh*'s efficiency, and we obtained it on every single test run.

On average, the POSIX-complaint *PaSh* with width 2 ran scripts 1.56 times faster running at most 2.29 times faster and at least 0.98 times as fast compared to running the script directly. On average, the *Bash*-complaint *PaSh* with width 2 ran scripts 1.58 times faster running at most 2.29 times faster and at least 0.97 times as fast compared to running the script directly. On average, the *Bash*-complaint *PaSh* with width 2 on scripts with small bashisms ran scripts 1.59 times faster running at most 2.31 times faster and at least 0.99 times as fast compared to running the script with the bashism directly.

On average, the POSIX-complaint *PaSh* with width 8 ran scripts 3.30 times faster running at most 6.65 times faster and at least 0.98 times as fast compared to running the script directly. On average, the *Bash*-complaint *PaSh* with width 8 ran scripts 3.29 times faster running at most 6.61 times faster and at least

0.99 times as fast compared to running the script directly. On average, the *Bash*-complaint *PaSh* with width 8 on scripts with small bashisms ran scripts 3.28 times faster running at most 6.43 times faster and at least 0.99 times as fast compared to running the script with the bashism directly.

Though we only used 10 benchmarks, we believe that these results are highly likely to extrapolate to most, if not all scripts. These tests aren't testing the speedups of the parallelization logic, they are just ensuring that the *Bash*-compliant *PaSh* is correctly exposing components of scripts as potentially parallelizable to *PaSh's* Just-in-Time engine. Since this is working correctly for 10 scripts there is no reason to believe that this will not work correctly in general.

Chapter 5

Limitations

As important as it is to share the success of this project, it is equally important to acknowledge the limitations of the *Bash*-compliant *PaSh*. In this chapter, we break down two types of limitations. The first, bugs, are issues with work completed in this project that should be fixed within the coming months and do not reflect how we expect the programs to work. It should be noted that all of the known bugs are bugs in programs that we are not responsible for maintaining. However, in all cases, we have reported the bugs to the appropriate maintainers and expect the bugs to be resolved in the next few months.

In contrast, inherent limitations are limitations that are expected and well-documented and can only be overcome by major changes to the structure of *PaSh*, or perhaps should not be fixed.

5.1 Bugs

In this section, we highlight three types of bugs. The first is bugs in the *bash* source code's command printing function, which affect the *libbash* printing function. These bugs, however, do not affect the functionality of *PaSh*. The second bug is in the *bash* source code's parsing function, which presents in the *libbash* parsing function. The third bug is in the *PaSh* infrastructure, independent of changes made as part of this project. Relevant links for these bugs can be found in the Appendix 8.

5.1.1 Bash Command Printing

Though this does not affect PaSh, there are a few bugs in the *libbash* printer that are worth discussing. The first bug appears in scripts for which the character `0x01` or `0x177` appear. In *bash*, users may put the literal integer value representing a character directly into a script rather than the character itself. There are a few characters that have a special meaning in the parser, though. However, if these characters literally appear in the script, the parser needs to escape the character to note that it does not have its usual special meaning, but rather, should be interpreted literally. Thus, if `0x01` or `0x177`, special characters, literally appear in a script, the Abstract Syntax Tree (AST) will contain `0x01 0x01` and `0x01 0x177` respectively. The first `0x01` specifies that the next character should be interpreted literally, and the second `0x01` or `0x177` is the literal character. The *bash* printer function, however, incorrectly does not unescape the `0x01`. Thus, when the AST is converted back to *bash* the extra character `0x01` is incorrectly written.

Another bug is uncovered during the printing of `coproc` commands. In *bash*, the `coproc` command allows commands to be run in the background, with a few extra special properties. In *bash*, there are three allowable syntax formats [24]:

```
coproc NAME compound-command
coproc compound-command
coproc simple-command
```

However, internally, every coproc is assigned a `NAME` with the default being `COPROC`. In the printer, a `coproc` command is printed as

```
coproc NAME command
```

which is valid if `command` is a `compound-command`, however, it is not valid if it is a `simple-command`. The printer incorrectly fails to distinguish between these two cases.

The final known bug in the *bash* printer appears in a special situation with the `case` command. The `case` command allows an expression to be matched to a series of patterns and code will be executed depending on which pattern is matched. For example, in the following script, if the value of the variable `ANIMAL` is `horse`, `dog`, or `cat`, `echo "four-legs"` is run, if it is `man` or `kangaroo`, `echo "two-legs"` is

run, and otherwise `echo "unknown"` is run.

```
case $ANIMAL in
  horse | dog | cat) echo "four-legs";;
  man | kangaroo ) echo "two-legs";;
  *) echo "unknown";;
esac
```

Critically, the `esac` keyword specifies the end of the `case` command. However, let's suppose that instead of matching `horse` we would like to match the word `esac`. It turns out that parsing this will fail, saying that it encountered an unexpected `esac`

```
case $ANIMAL in
  esac | dog | cat) echo "four-legs";;
  man | kangaroo ) echo "two-legs";;
  *) echo "unknown";;
esac
```

However, per the `case` command syntax specification, a user may put a left parenthesis before any pattern, so this is valid syntax:

```
case $ANIMAL in
  (esac | dog | cat) echo "four-legs";;
  man | kangaroo ) echo "two-legs";;
  *) echo "unknown";;
esac
```

In any other case, the choice to include the left parenthesis is just stylistic, however, when `esac` is being matched this matters. The *bash* printer, however, does not consider this case and will always omit the left parenthesis.

5.1.2 Bash Parsing

The bug with `esac` in case commands also appears during script parsing. Consider the following script:

```
echo $(case esac in (esac) echo hello;; esac)
```

Here, we have a case command within a command substitution. However, during parsing, the left parenthesis before somehow gets removed, and the word that should be

```
$(case esac in \n (esac)\n echo esac\n ;;\nesac)
```

appears as

```
$(case esac in \n esac)\n echo esac\n ;;\nesac)
```

In general, commands inside of command substitutions appear to be stylistically formatted when they appear in the AST, so this may be the same bug that appears in the *bash* printer, however, in this case, it presents in the parser.

5.1.3 Setting IFS

In *bash*, the variable `IFS` represents the internal field separator, which determines how *bash* separates words when separating strings. For example, consider the following script:

```
IFS=':'
word="hello:world"
for i in $word
do
    echo $i
done
```

In this case `hello` and `world` we be printed in two separate iterations of the loop since they are separated by `IFS`. However, if we did not set `IFS`, `hello:world` would be printed as one word. When `IFS` is set in a script and the script is run with *PaSh*, however, *PaSh* crashes. Since the same error happens regardless of whether *bash* mode is used, this is considered to be a bug in *PaSh*'s existing infrastructure.

5.2 Inherent Limitations

In this section, we describe two limitations inherent to the *Bash*-compliant *PaSh*. The first limitation is that scripts with commands that change how *bash*'s parsing

logic work are not guaranteed to be parsed correctly by *libbash*. The second limitation is that scripts that reference *bash*'s internal variables are not guaranteed to run and have the same results with and without *PaSh*.

5.2.1 Commands that Change Parsing

In *bash*, there are several commands that, if executed, change how parsing works. For *bash*, this is fine because commands are parsed and executed line-by-line. However, this presents issues for *PaSh* and *libbash*, which parse an entire script before executing any of it. Consider, for example, aliases in *bash*. Aliases are essentially direct mappings from one word to another word. Aliases are similar to variables, however, they differ in that they are considered during parsing time, and they can also change what is syntactically correct in a script. Consider the following script:

```
shopt -s expand_aliases
alias my_command="echo‘"
my_command hi’
```

The `shopt -s expand_aliases` tells *bash* to expand aliases in the script. `my_command` aliases to `echo ‘` which makes the entire last command `echo ‘hi’`. However, if this alias was not set, this script would not even parse, complaining that there is an unclosed single quote in the script. If *libbash* attempted to parse this script it would fail for this reason—it did not actually execute the command `shopt -s expand_aliases` or `alias my_command="echo ‘"`.

While a viable workaround may seem to be to execute only `shopt` commands and `alias` definitions, a different example will illustrate that this is not always possible:

```
shopt -s expand_aliases
if some_complex_command; then
    alias my_command="echo‘"
fi
my_command hi’
```

In this script, we cannot determine whether the `alias` is set without actually executing `some_complex_command`. `Aliases` and other similar constructs cannot always be determined without fully executing a script, which defeats the purpose of *libbash* in the first place. Though it can be determined in many simpler situations, for consistency we chose to completely ignore such constructs.

Other examples of similar commands that change how a script is parsed include `shopt -s extglob` which was described earlier in the dissertation in Section 4.1.3 and `unset` which removes aliases and similar constructs.

5.2.2 Bash Variables

In *bash*, there is a group of variables that are set during the runtime of a script that are set to capture details about the execution environment. Since *PaSh* must inherently execute a script in a different environment, these variables will look different depending on if a script is run with *PaSh*. Examples of these variables include `BASH_LINENO`, which stores information about the line numbers where functions were called, and `HISTCMD` which stores the index of the current command in the history list of commands executed.

Chapter 6

Future Work

In this chapter, we discuss three topics that could act as areas of future work in this field. The first is determining expansions natively in the function which converts a *libbash* Abstract Syntax Tree (AST) to a *shasta* AST, rather than invoking an `echo` shell in *PaSh*. The second area of work is modifying either *libbash* or *PaSh* to creatively approach `aliases`, the `shopt` command, and other similar utilities that change how *bash* parses. The final area of work is extending *PaSh* to work with other *shell* implementations.

6.1 Native Expansions in Libbash to Shasta

In *libbash*, the fundamental unit of text is a `word`, a string of characters usually separated from other words by spaces. The *bash* parser makes no distinction between different types of words beyond flags which hint to the execution function what types of words they might be. For example, consider three words in *bash*:

```
some_word
$var
${param:+word}
```

These are all just words in *bash*, however, they all mean different things. `some_word` is literally just the string `some_word`. `$var` means that the variable `var` should be expanded and replace `$var`. `${param:+word}` means that if `param` is not assigned as a variable this should be expanded to an empty string, otherwise, it should be expanded to `word`. While *libbash* does not treat these cases

differently, *libdash* and *shasta* do.

Libdash's basic unit is the `ArgChar`. In most cases, this is just a character. So for example, `some_word` would be represented as a list of characters that make up the word. However, `ArgChar`s also represents units of expansion. So, for example, `$var` is represented by a `VArgChar` with a field containing the list of characters that make up `var`. Neither approach is more or less correct, however, *libdash*'s approach does provide more detail at parsing time which *bash* chooses to process at execution time. However, since *PaSh* uses *shasta*, *PaSh* relies on the assumption that all expansions are identified at parsing time.

Implementing a function that determines all of the expansions in a *libbash* AST is certainly doable, and was considered as part of this project, however, we determined that it would be out of scope given the timeline. The function in *bash* that does this work during command execution is several thousand lines of code, and implementing it in *python* will be a long and tedious process.

For the time being, we implement a workaround for not doing this expansion analysis that we describe in Section 3.3.2. However, this workaround does have additional overhead which is relatively trivial (less than a few hundred milliseconds in most cases), but it is not ideal, especially for the moral correctness of *shasta*. A great area for future work would be to implement this expansion logic in *python* and use it when converting scripts from *libbash* to *shasta*.

6.2 Handling Commands That Change How Parsing Works

As we discussed in Section 5.2.1, there is a fundamental limitation when parsing *bash* scripts, because some commands in *bash* change how parsing is done for the remainder of the script. Though it is not possible to implement a completely correct *bash* parser that does not also execute scripts, there are ways that we could get closer to correctness in *libbash* and *PaSh*.

One approach could be to execute these commands only if they are at the top level of a *bash* script, because in practice, people usually use these commands at the top level. This is by no means an easy task, because bash supports several

different commands that change parsing in subtle ways, however, it would be a step towards correctness on a larger set of scripts. This could certainly be an area of future work in either *libbash* or *PaSh* that could allow users to use more scripts with these tools.

6.3 PaSh Compliance with Other Shells

Although *bash* is the most popular *shell* interpreter on *Linux*, there are several other popular *shell* implementations such as *zsh* and *fish* that have features that do not exist in *bash*. Adding the ability to run scripts in these *shell* implementations with *PaSh* could be valuable for users of these *shell* interpreters.

Chapter 7

Conclusions

PaSh is a tool used to speed up the execution of POSIX *shell* scripts. *PaSh* has proven to be popular, receiving hundreds of stars on Github. Despite its success, *PaSh* is severely limited by only being able to run POSIX *shell* scripts. In practice, most people do not write POSIX *shell* scripts because POSIX is not actually a *shell* implementation — it is a standard specification that all *shell* implementations are expected to follow. In practice, most people write *shell* scripts for the *shell* implementation they use, which on Linux systems is usually *bash*. The goal of using *PaSh* on a *shell* script is to speed up execution, so requiring the user to modify their *shell* script to be *POSIX*-compliant defeats the purpose.

The goal of this project was to upgrade *PaSh*, a popular tool for parallelizing *shell* scripts, to support *bash* scripts. To complete this project, we first implemented a *bash* parser in *python*. Next, we changed *shasta*, *PaSh*'s AST interface to support additional features that *bash* supports. Finally, we integrated these changes into *PaSh* so that now, a user can pass in a `--bash` flag when running *PaSh* and it will be parsed as a *bash* script.

We hope that adding this feature greatly expands the audience that *PaSh* reaches and helps. We also expect that this addition will open up more research opportunities in *PaSh*, since the types of scripts *PaSh* works on is now significantly expanded. Finally, we envision several areas of future work that will make this system even more valuable for users.

References

(1) Octoverse: The state of open source and rise of AI in 2023, https://github.blog/2023-11-08-the-state-of-open-source-and-ai/, Accessed: 2024-05-02.

(2) A Guide to Posix — Baeldung on Linux, https://www.baeldung.com/linux/posix, Accessed: 2024-04-22.

(3) The Open Group Base Specifications Issue 7, 2018 edition, https://pubs.opengroup.org/onlinepubs/9699919799, Accessed: 2024-04-22.

(4) Top 5 open source command shells for Linux, https://opensource.com/business/16/3/top-linux-shells, Accessed: 2024-04-22.

(5) 10 Most Popular Open Source Linux Shells, https://tecadmin.net/most-popular-open-source-linux-shells/, Accessed: 2024-04-22.

(6) Vasilakis, N.; Kallas, K.; Mamouras, K.; Benetopoulos, A.; Cvetković, L. *Proceedings of the Sixteenth European Conference on Computer Systems* **2021**, 49–66.

(7) PaSh: Light-touch Data-Parallel Shell Processing — Github, https://www.linuxfoundation.org/press/press-release/linux-foundation-to-host-the-pash-project-accelerating-shell-scripting-with-automated-parallelization-for-industrial-use-cases, Accessed: 2024-04-22.

(8) Linux Foundation to Host the PaSh Project, Accelerating Shell Scripting with Automated Parallelization for Industrial Use Cases, https://www.linuxfoundation.org/press/press-release/linux-foundation-to-host-the-pash-project-accelerating-shell-scripting-with-automated-parallelization-for-industrial-use-cases, Accessed: 2024-04-22.

(9) The birth of the Bash shell, https://opensource.com/19/9/command-line-heroes-bash, Accessed: 2024-04-23.

(10) Redirections | Bash, https://www.gnu.org/software/bash/manual/html_node/Redirections.html, Accessed: 2024-04-23.

(11) Pipelines | Bash, https://www.gnu.org/software/bash/manual/html_node/Pipelines.html, Accessed: 2024-04-23.

(12) Shell Command Language, https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html, Accessed: 2024-04-23.

(13) How to Change Default Shell in Linux — Geeks for Geeks, https://www.geeksforgeeks.org/how-to-change-default-shell-in-linux/, Accessed: 2024-04-23.

(14) Bash Specific Features, https://web.mit.edu/gnu/doc/html/features_4.html, Accessed: 2024-04-23.

(15) Bash Features (Bash Reference Manual), https://www.gnu.org/software/bash/manual/html_node/Bash-Features.html, Accessed: 2024-04-23.

(16) libdash | Github, https://github.com/binpash/libdash, Accessed: 2024-04-24.

(17) shasta | Github, https://github.com/binpash/shasta/tree/main, Accessed: 2024-04-24.

(18) libbash | Github, https://github.com/binpash/libbash/tree/main, Accessed: 2024-04-25.

(19) libbash | PyPI, https://pypi.org/project/libbash/, Accessed: 2024-04-25.

(20) bash.git, https://git.savannah.gnu.org/cgit/bash.git/, Accessed: 2024-04-25.

(21) Johnson, S. C. Yacc: Yet Another Compiler-Compiler, https://www.cs.utexas.edu/users/novak/yaccpaper.htm, Accessed: 2024-04-26.

(22) ctypes — A foreign function library for Python, https://docs.python.org/3/library/ctypes.html, Accessed: 2024-04-28.

(23) Setting IFS crashes PaSh | Issue 718, https://github.com/binpash/pash/issues/718, Accessed: 2024-04-29.

(24) Coprocesses, https://www.gnu.org/software/bash/manual/html_node/Coprocesses.html, Accessed: 2024-04-29.

(25) Conditional Constructs, https://www.gnu.org/software/bash/manual/html_node/Conditional-Constructs.html, Accessed: 2024-04-29.

(26) How to make bash scripts work in dash, https://mywiki.wooledge.org/Bashism, Accessed: 2024-05-04.

(27) print_cmd.c - line 379 - bash.git - bash, https://git.savannah.gnu.org/cgit/bash.git/tree/print_cmd.c#n379, Accessed: 2024-05-07.

(28) fix for dequoting words in pretty-print mode; posix mode changes for read-only/export invalid identifier errors - bash.git - bash, https://git.savannah.gnu.org/cgit/bash.git/commit/?h=devel&id=fa0b002927c2897f80da762dd4196d688a46a3ab, Accessed: 2024-05-07.

(29) print_cmd.c - line 359 - bash.git - bash, https://git.savannah.gnu.org/cgit/bash.git/tree/print_cmd.c#n359, Accessed: 2024-05-07.

(30) fix history expansion to not perform quick substitution on a new line that's part of a quoted string; save the value of $_ around prompt string decoding - bash.git - bash, https://git.savannah.gnu.org/cgit/bash.git/commit/?h=devel&id=aa2d23cfac90bebe2924ba075fef0a03fddd521d, Accessed: 2024-05-07.

(31) print_cmd.c - line 761 - bash.git - bash, https://git.savannah.gnu.org/cgit/bash.git/tree/print_cmd.c#n761, Accessed: 2024-05-07.

(32) fix for printing case pattern lists beginning with “esac”; several fixes for expansion when IFS contains DEL - bash.git - bash, https://git.savannah.gnu.org/cgit/bash.git/commit/?h=devel&id=54f3ed2278025081f897b9bd958fcf099fd5be18, Accessed: 2024-05-07.

Chapter 8

Appendix

8.1 Known Limitations Table

Known Limitations		
Limitation	**Relevant Code**	**Bug Report/Fix**
bash issue printing words with the internal escape character	[27]	[28]
bash issue printing coproc commands	[29]	[30]
bash issue printing case commands with esac as a pattern	[31]	[32]
PaSh issue printing case commands with esac as a pattern	N/A	[23]