Pomona College
Department of Computer Science

Mechanized Semantics for Word Expansion in the POSIX Shell

Austin Blatt

April 29, 2018

Submitted as part of the senior exercise for the degree of
Bachelor of Arts in Computer Science

Professor Michael Greenberg, advisor

Copyright © 2018 Austin Blatt

The author grants Pomona College the nonexclusive right to make this work available for noncommercial, educational purposes, provided that this copyright statement appears on the reproduced materials and notice is given that the copying is by permission of the author. To disseminate otherwise or to republish requires written permission from the author.

Abstract

The POSIX Shell is the default command line interface with which computer scientists interact. This paper describes the creation of an executable semantics for word expansion in the POSIX Shell. Word expansion is the first of two steps in the evaluation of a POSIX shell command. By developing a model of its operational semantics we formalize its execution. This formalization furthers our understanding of how the POSIX Shell operates, and is a step in the direction to creating a full formal model of the POSIX Shell. Our executable semantics were developed in Lem, which can be compiled to Coq for proofs and OCaml for execution.

Contents

Abstract i

1 Introduction **1**
- 1.1 The POSIX Shell 1
- 1.2 Lem 2

2 POSIX Shell Overview **5**
- 2.1 Interesting Edge Cases 8
- 2.2 Bugs! 9

3 POSIX Word Expansion Semantics **11**
- 3.1 Word Expansion 11
- 3.2 Symbolic 14

4 Mechanized Semantics **17**

5 Related Work **21**
- 5.1 Background 21
- 5.2 Defining the Operational Semantics 23
- 5.3 Modelling Memory 29
- 5.4 Comparison 30

6 Future Work and Conclusion **31**
- 6.1 Future Work 31
- 6.2 Conclusion 32

Bibliography **33**

Chapter 1

Introduction

1.1 The POSIX Shell

The POSIX shell ("the Shell") is the standard, and most popular, command-line interface by a large margin. A command-line interface is an interactive means of controlling the file system and processes of a computer by using various utilities such as `mv` and `cp` for moving and copying files respectively. The POSIX shell is used for a wide range of tasks including an interactive command-line interface, shell scripts, and even system start up scripts for operating systems. As such, the Shell controls critical pieces of software and is crucial to the operation of a computer. Other shells exist, such as `zsh`, but our model focuses on the POSIX specification due to its prevalence.

Unlike common programming language, the POSIX shell specifies no real memory model apart from simple shell variables. The main "data" that the POSIX shell interacts with is the *file system*. This makes it inherently dangerous. In a standard programming language, errors only damage data in memory (unless you then write that data to disk). In many cases this is the desirable use case because in-memory data types can be much more complex than the shell's data type, which can only represent strings. But at some point a programmer may need more direct access to system data and processes. In that case, the programmer needs to access the computer via the Shell. As we'll discuss further in Chapter 2, some of the features of the POSIX shell have edge cases that can have permanent, damaging effects on the file system.

In order to investigate the semantics of the shell we created a mechanized small-step semantic model of shell expansion. Our long-term goal is to create an executable operational semantics for the Shell, and this paper describes

the semantic model of Shell expansion. As described in Chapter 6, we need to formalize the process of Shell evaluation as well before it will be a full model of the Shell semantics.

The process of creating a semantic model of a programming language, including the Shell, is an important undertaking. Very few users of the Shell will interact with the Shell specification. It is a dense prose, and if you are not intimately familiar with the organization of the specification, a piece of important information relating to your use case could be located essentially anywhere. This leads to the strategy of learn by example, which will take less time than a reading of the specification. But the specification is still the ground truth, and an operational semantics can aid in our understanding of a language and the addition of new features (or the modification of existing ones). The semantic model provides a basis for using proof assistants to verify language features [Kre15]. Krebbers' work in creating an operational semantics for C, semantic models of other languages, and a short history of attempts at formalizing programming language specification is given in Chapter 5.

The main contribution of our work is an executable semantic model of Shell expansion. This model was developed from a close reading of the POSIX standard, which is mostly prose that mixes a description of parsing, evaluation and definitions of utilities such as `mv` and `rm`. Our methodology of modelling the POSIX shell is described in Chapter 3.

1.2 Lem

Lem is a language developed by Peter Sewell intended to simplify writing large-scale semantic models [OBZNS11]. The language can be compiled into Ocaml (an executable functional programming language), or Coq, and Isabelle/HOL (theorem provers). We wrote the semantics for the POSIX shell in LEM for the potential to integrate with SibylFS, a model of the POSIX file system also developed by Peter Sewell [RST+15]. By integrating with this model, we would present a full model of the POSIX standard, from file system APIs to command-line interface. As of now, our model implements a simplified model of the file system.

Using LEM, we have developed a mechanized semantics for POSIX shell word expansion. Mechanized semantics are useful in their ability to quickly determine "how does this program execute?" and "what does this program do?". For a human, these are challenging questions to answer. Take a look at a simple example for the POSIX shell, `echo $x`. Most people familiar with

the shell will be able to tell you what this program does, it prints the value of `x` to standard output. For larger shell-scripts, it will take careful analysis of the code regardless of the individual's prior experience with POSIX. The question of how this program does what it does is more subtle. The execution of this program depends on, at a minimum, three different sections of the POSIX standard. Shell Introduction (2.1) to determine the order of evaluation, Word Expansion (2.6) to determine how $x is expanded, and Shell Commands (2.9) to determine how the `echo` utility operates [IG18]. These sections are written in English prose that has to be interpreted in order to determine how the command evaluates. By developing a mechanized semantics of this process, we have created a tool that will quickly compute the answer to both questions.

The development of a mechanized semantics is also a useful exercise to determine where differing implementations have interpreted the standard's prose in differing ways, and to find examples where implementations have made mistakes in their interpretations. We found a bug (Section 2.2) in a widely-used shell implementation in just this manner.

Chapter 2

POSIX Shell Overview

The following examples are intended as an overview of word expansion, so the information in this section can be found in the POSIX specification (Section 2.6 Word Expansions) [IG18]. At the highest level view, the Shell is a program that takes in a list of strings, which the Shell calls Words, and evaluates a command. As an example, `rm file1` will be parsed by the Shell as a list of two strings `rm` and `file1`. The first string will be interpreted as the command to execute and the remaining strings are its arguments. `rm` is a built-in utility that, given the argument `file1`, will remove `file1` if it exists in the current working directory. There are actually many ways to remove that file, `x="file1"; rm $x` will remove that same file, and so will `x=0; rm file$((x=x+1))`. In fact, even ``rm fi`echo le1` `` will remove `file1`. In all three cases, we have succeeded in removing the file, so what's different about these various ways of removing `file1`?

The second command shown, `x="file1"; rm $x`, removes the file through *parameter expansion*. Parameter expansion is the process by which the Shell expands a variable into its value. In this command it is simple, `x` will expand directly to `file1` and the `rm` utility will remove it. It is possible to format a parameter expansion to control its output. The command `rm ${x="file1"}` *MAY* remove `file1`. If x is already set to another value, the Shell will attempt to remove that file instead, but if `x` is unset, this will remove `file1` and the end state will be identical to the original command where the file `file1` is removed and `x` has the value `file1`.

The next example above, `x=0; rm file$((x=x+1))`, removes the file using *arithmetic expansion*. First, `x` is assigned to zero, but keep in mind that this is the string `"0"`. Then, prior to executing the command, arithmetic expansion must occur on `$((x=x+1))`. Since `x` was just set to 0 in the Shell, this arithmetic expansion will evaluate, assign the result of the arithmetic

evaluation (in this case 1) to `x` and substitute 1 into its place in the original command. Therefore, this command will also execute as `rm file1`. Note that the end state is slightly different, the shell variable `x` is set to 1 instead of `file1`. This could be useful if we had another file, `file2`, that we wanted to remove next. Another call to `rm file$((x=x+1))` would remove that file.

In our final and least clear example, ``rm fi`echo le1` ``, we have used *command substitution* to remove the file. Command substitution allows a command to be nested inside of another command. The command `echo le1` will print the string `le1` to standard out, and by the specification of command substitution, the *control code* (everything between and including the backticks), will be removed and replaced with the standard output of the command. Therefore, the command we end up running is `rm file1` just like all the other examples, but no variables have been set.

If `file1` exists at your user's home directory, it can be removed with `rm ~/file1`. This is the forth, and final, example of control code expansion, *tilde expansion*. There are two cases of tilde expansion, `~` and `~username`. In the first case, the tilde character will expand to the value of the `HOME` shell variable (so it is equivalent to the parameter expansion `$HOME`). In the second case, the Shell will look up the home directory of `username` in the user database. In my shell history, tilde expansion is most commonly used to reload my Bash configuration file with `source ~/.bash_profile`. A close second is `cd ~`, but in this case the tilde is entirely irrelevant. According to the specification of the `cd` utility, *"[i]f no directory operand is given and the `HOME` environment variable is set to a non-empty value, the `cd` utility shall behave as if the directory named in the `HOME` environment variable was specified as the directory operand"*. This means that the following three commands are equivalent `cd`, `cd ~`, and `cd $HOME`.

The four previous commands constitute *control code expansion*. A control code is a string input to the shell that is delimited by special characters (e.g. the dollar-sign `$`, backticks `` ` ``, and the tilde `~`) that allows the shell to identify it as something that needs to be subject to expansion. After a control code has been expanded, there are three additional stages of expansion that it can go through. These stages will finalize the results of expansion as fields, expand patterns to match file names, and remove quotes.

First, *field splitting* will look at the strings that were a result of expanding control codes for `IFS` characters, and create separate fields from strings that are delimited by an `IFS` character. `IFS` is a special shell variable that contains the *internal field separator* characters. By default, or if the `IFS` variable is unset, whitespace will result in field splitting (e.g. *space*, *tab*, and *newline* characters). Since field splitting only occurs on the results of

control code expansions, in the above example, `x="file1"; rm $x`, after `$x` has been expanded to `file1` field splitting will occur on the string `file1`, but will not occur on `rm`. The string, `file1`, does not contain any `IFS` characters, so the final result is two fields `rm` and `file1`. However, it would be possible for that parameter expansion to result in more than one field, even though it contains no whitespace, if `IFS` was set to a string containing a character found in `file1` (e.g. `IFS=e` would result in three fields `rm`, `fil`, and `1` with the `e` removed because it was consumed as a field separator).

But wait, how did `rm` and `fil` become two separate fields when *space* was no longer a field separator? The short answer, the field separator between `rm` and `$x` is created during *parsing*, which is something our model does not attempt to describe. The long answer, as quoted from section 2.3 (Token Recognition) of the POSIX specification, which contains 10 rules for token recognition in parsing, *"7. If the current character is an unquoted <blank>, any token containing the previous character is delimited and the current character shall be discarded."* I include that, not because anyone should read that once and come away with a feeling of enlightenment, but to highlight the difficulty in understanding the prose in the POSIX specification. A few questions might come to mind after reading that, what is a <blank>? Those unfamiliar with parsing a programming language may also wonder, what does is mean to delimit a token? And why are we discarding the current character? The Shell specification does not clarify this, but a <blank> is a *space* or *tab* character. They use this to differentiate the two from *newline* characters in regards to parsing comments. According to this rule, during parsing of the command, `rm` and `$x` will be identified as two separate tokens and the *space* character between them will be discarded. The result for Shell expansion, and our model, is that it will start with two separate words and even changing the `IFS` won't effect the final fields generated.

After field splitting, pathname expansion, which also occurs on unexpanded strings, expands special characters such as `?` and `*` to valid file names. Pattern matching occurs via a regular expression syntax described in the POSIX specification, with a few filesystem-related caveats. The pattern `*`, in most regular expressions, matches anything. But if you try it out in the Shell, it will only match the files in the current directory. Pathname expansion requires two characters to be matched literally, a <slash> character must always be matched with a literal slash, and if a file name begins with a <period>, the period must be matched literally. If pathname expansion does not match a file, the string that was entered as the patten is left in the command without change. If pathname expansion finds files that match the pattern given, it replaces the pattern with the files that matched, with

each file being its own field. For example, `mv * ../dir/` will expand to $n + 2$ fields, where n is the number of files in the directory (that don't begin with a <period>). In this respect, the `*` character is the same as typing out all the filenames in the current directory, just less tedious.

Quote removal is the final stage of expansion. Unlike standard programming languages where quotes delimit strings, quotes remain a part of the string until quote removal. In Shell expansion, quotes control field splitting. They are also used to prevent the expansion of characters `~` , `*`, and `?`, but this is not encapsulated by our model because it is handled during parsing. In the previous section, we described that `rm $x` will be parsed as two words, but could expand to more than two fields. If we instead wrote `rm "$x"`, we can now be sure that the final result of expansion is two fields, `rm` and the result of expanding `$x`. This is because field splitting will not occur on an expansion inside quotes. After this stage, quotes are removed. This has some interesting edge cases that we will discuss in the next section.

2.1 Interesting Edge Cases

The edge-cases that we will go over here are a result of field separation mistakes. Strings we intended to act as a single argument will be interpreted as two. Errors such as this arise as a side effect of field splitting because field splitting must be handled at every use site of control code expansion. This means that when assigning to a variable, you cannot say "this should always be one field". If I assign `x="a b"`, and I am intending for *every* use case of this string to be a single field, any time I expand `$x` I must remember to quote it (`"$x"`) or it would expand to two fields.

This can lead to extraneous deletions when used with the `rm` utility we've been using as an example. Let's imagine a directory with three files `file1`, `file2`, and `file1 file2`. So we have a file that we've named with a *space* character in it. Most computer scientists will know this is a bad idea, but let's find out why. Since I know I shouldn't have a file named that, I'd like to delete it. In my shell environment, I have a variable `x` that was set by the command `x="file1 file2"`. So I run `rm $x` to delete it. The command will execute and terminate without error. Much to my dismay, when I look at the remaining files, I realize that not only did I fail to delete the file I intended to, but I deleted *both* files that I wanted to keep. The only file remaining in the directory is `file1 file2`.

Let's dive into what went wrong. First, when I assigned `file1 file2` to `x`, I quoted the string so that the space would not split the string into

two fields during assignment. Assignments can only be given a single word, a second field would be interpreted as a command name. But when I went to use `x`, I failed to quote its expansion. Therefore `$x` expanded to `file1 file2` and during field splitting this string was separated into two fields, `file1` and `file2`. The result was that the `rm` utility was asked to remove two files, one named `file1` and the other named `file2`. Both files existed, so the `rm` utility removed those files and completed "successfully". If I had quoted correctly, e.g. `rm "$x"`, the `rm` utility would receive a single argument, `file1 file2`, and remove the file I had intended to delete.

A similar edge case can be encountered with pathname expansion and the `mv` utility. You might run, the command `mv * dir` to move files in the current directory into another, but imagine for a second you ran the command `mv *`. If you happen to be a professional writer of shell scripts and don't think you'd ever make such a mistake as this, imagine you were instructing someone on the command they should run, and they left off the directory. Most of the time this will error, but sometimes this will succeed and in some cases it will result in file(s) being deleted. This command will run to completion in two cases. First, it will succeed if there are two files in your directory, but it will delete one of the files. Let's say you have two files named `a` and `b` and you execute `mv *`. Pathname expansion will expand `*` to two fields, and the resulting command that is executed is `mv a b`. This is the command that "moves" `a` to `b`. In this process you renamed the file `a` and lost your file reference to `b`, so you will find it quite difficult to retrieve the file and its contents (even though the file may still reside on disk).

This command will also evaluate and succeed in a second case, when there is at least one file and one directory in the current directory and in sorted order (specified by your Shell locale) the last item is a directory. In this case, it will move all files and directories in the current directory into the last directory and may or may not have deleted any files. You will only delete files if one of the files or directories that was moved into the last directory has the same name as a file in the last directory. The move utility performs no checks on overwriting already-existing files, so it is imperative that you are careful when moving files.

2.2 Bugs!

In the process of creating our mechanized semantics, we also found a bug in a widely used shell, Dash. Finding bugs in existing implementations is a notable contribution that will occur as a result of creating mechanized

semantics. As has been shown in previous sections, the POSIX specification is dense prose that is not always clear. It is also long. As published in IEEE Std 1003.1-2017, the Shell specification is over 1000 lines long [IG18]. There are only three groups that will read the specification in detail, the authors, developers of shells such as Bash and Dash, and researchers creating mechanized semantics.

In an attempt to create a semantics of a language (or a compiler for that language), a detailed reading of the specification is required. In our process we consulted the POSIX specification before writing any code, if a feature was undefined or we wanted to test its behavior we would verify our understanding of the specification against two Shell implementations, Bash and Dash. This meant that we were not only reading portions of the POSIX specification closely, but we were formulating the specification into small-step operational semantics while testing the functionality of two implementations. This resulted in a fairly detailed testing of both shells against the specification focusing initially on simple cases and gradually moving to edge cases and exceptions to certain rules. This may not be the workflow of everyone who extracts operational semantics from a prose specification, but it seems a fairly standard, and logical, workflow. As we found out when we found the bug in Dash, it is important not to assume the existing implementations are correct and always fall back on the specification as the ground truth in order to develop an accurate model.

Dash is a shell that was developed to speed up the execution of shell scripts used in the process of Debian start-up. Dash claims that it is roughly four times faster than Bash, and it became the default script execution shell for Ubuntu, in 2006, and Debian, in 2014, and subsequently all of the operating systems built off Debian as they upgrade to Debian 6.0 [Deb]. Apart from a speed increase, one of the main goals in developing Dash was a more POSIX-compliant shell than Bash.

The bug in Dash is in regards to its handling of arithmetic expansion. The command we will run is `echo $((x=x+2))`. In general, this command will add 2 to x and then print that new value of `x` to standard output. This command will fail if `x` is not set to a numerical value (you can't add 2 to the string `foo`). But what will happen when `x` is unset? The specification says that `x` should default to 0, so the result of the command above would be 2. The result in dash is an error (exit code 2) and the error message `Illegal number`.

Chapter 3

POSIX Word Expansion Semantics

The POSIX standard makes a clear distinction between its two phases of execution, *word expansion* and *command evaluation*. Word expansion's goal is to turn user inputs into fields. With the user inputs expanded into their final state, the shell then evaluates the given command. At this point, our shell model for command evaluation is incomplete so we will treat commands abstractly.

3.1 Word Expansion

The first phase of the Shell is *word expansion*, which aims to turn user input, Words, into Fields that will be used in evaluation. The types used by our model are described in Figure 3.2 below. In general, POSIX shells perform the seven steps of *word expansion* in a single left to right pass over the input string. You've seen examples of each of these steps in Chapter 2. Our model is built off the same seven steps of word expansion, but the steps are performed in four stages (see Figure 3.1). The first four cases describe control code expansion, and the following three are field splitting, pathname expansion, and quote removal. This four stage model, which is diagrammed in Figure 3.1, is useful to visualize because it forms the basis for the type system specified in Figure 3.2 that the rest of this chapter will be spent explaining.

In order to model *word expansion*, we represent the POSIX Shell AST using the types specified in Figure 3.2. Word expansion is a process that expands Words into Fields. We use the Dash parser to obtain words, which

can be one of five things. Words, at their simplest, are a string followed by more words, $\mathtt{s}; ws$. These words come separated by initial *field separators*, ␣, which are the initial `space` and `tab` characters present at parsing. Words that come out of the Dash parser may also be *control codes*, represented with k. The fifth type represents a symbolic word, λ. These are not part of the POSIX shell or normal shell implementations, but this type is used in our semantics in order to represent an undetermined result (e.g. the result of a command).

3.1.1 Control Codes

The interesting type of a *word* is a *control code*, which represents something that needs to be expanded before evaluation. First we have types that represent *tilde expansion*, `~` and `~s`. A single `~` will expand to the current users home directory (e.g. `/Users/username`) while `~s` will expand to the home directory of the user `s` ((e.g. `~ root` expands to `/var/root`).

Next, we see $\mathtt{\$\{s} \mid \phi\}$, which is the type representing a paramter expansion where `s` is the name of the shell variable to lookup and ϕ is one of a number of lookup formats that provide a way to control the result when `s` is set, set to null, or unset. The default format, $\bullet$, will lookup the value of `s` in the shell environment and substitute its value, or the *null* string if the variable is not set. If you'd like to specify a default value, $\mathtt{-}w$ will substitute the value of `s` in the shell environment, if it exists, otherwise it will substitute w. The $\mathtt{:-}w$ format is identical to the prior default value format, but it is the *null* default format, so it will substitute w if `s` if unset, or if `s` is set to the *null* (emtpy) string. Three other format pairs follow the same structure as the previous pair of formats. First $\mathtt{=}w$ is the assign format, which will assign the result of expanding w to `s` if `s` is unset, whereas $\mathtt{:=}w$ will assign w to `s` if `s` is unset or it is set to *null*. Otherwise, both return the value of `s`. Next we have a pair of error formats, $\mathtt{?}w$ and $\mathtt{:?}w$. Following the pattern of the previous two pairs, $\mathtt{?}w$ will error (and exit) if `s` is unset and $\mathtt{:?}w$ will error (and exit) if `s` is unset or set to *null*. The final pair functions slightly differently, they are the alternate value ($\mathtt{+}w$) and *null* alternate value ($\mathtt{:+}w$) formats. The alternate value format will substitute w unless `s` is unset, when it will substitute *null*. The null alternate value operates similarly, but it will only substitute w if `s` if set and not *null*. The `#` format will return the numerical value of the length of `s` without performing any other expansion, if `s` is unset, it returns 0 as a string.

The final four parameter expansion formats are shortest and longest prefix and suffix formats. All prefix and suffix operators take a *word* argument

w that is first expanded, which could even result in nested parameter expansion. The result is then interpreted as a pattern that is matched against the value of `s` in the environment. That matching part of the value, if any exists, is removed and the remaining is substituted in as the result. Two suffix matches exist, `%`w will match the shortest suffix and `%%`w will match the longest suffix. Two similar formats exist for `#`w, the shortest prefix, and `##`w, the longest prefix. Let's take the example of, `s` = "`aabb`". If the format used is `%b*`, which matches zero or more `b`'s, the resulting string is `aab`. If the longest suffix operator, `%%`w had been used instead, it would match and remove both `b`'s. The same is true for the shortest and longest suffix formats when used with the pattern `a*`.

Not all outputs will exist in shell variables before you want to substitute it in to your current command, for that reason the shell provides the ability to nest commands `$(`ws`)` is also provided. The command is executed in a sub-shell and the result (standard output) is substituted into the current command. In order to model this functionality in full, we would need a model of command evaluation, so for now it is left abstract and its return value represented by the symbolic result, λ.

In our model, there are three control code types that are not part of the POSIX specification. These paramters are used when further expansion inside a paramter expansion is necessary before performing the action specified by the parameter expansion format such as assignment, pattern matching, or arithmetic expansion. For example, if a parameter expansion such as `${s` | `:=`w`}` results in an assignment to `s`, the original parameter type is replaced, in its entirety, by `${s` *assign expws*`;`*ws*`}` . The *word*, w, is expanded in incremental steps to an *expanded word* `e`. Once all of w has been expanded, it can assigned to `s`. For a similar purpose, our model uses `${s` *match* **ss sm** *expws*`;`*ws*`}` to expand w into a pattern before it will be matched against the value of `s`. The *substring side*, either prefix or suffix, is remembered with `ss`. This runtime technicality also remembers whether or not it will find the the shortest or the longest possible matching present in the *substring mode*, `sm`. The final special control code used in our model is created when `${`x | `?`w`}` or `${`x | `:?`w`}` trigger an error. Before the program can error and exit, it must expand w and print the result to standard out.

3.1.2 Expanded Words

The result of expanding control codes is *expanded words*. This is an intermediate type used by the model to keep a record of important pieces of

information for the final three steps of word expansion. First, field splitting will need to know which strings are the result of a control code expansion as described above. Field splitting will not occur on a string unless it is the result of a control code expansion, therefore control code expansions as described in the previous section 3.1.1 generate `exp s` for types where a string type can be found and λ when the control code was a command, which produces an unknown (*symbolic*) result. The *symbolic* type represents a unknown value due to commands being left abstract. Quoted strings are also exempted from field splitting, which is tracked by `"s"`. Strings that are not a result of control code expansion, `s` and existing field separators ␣, are unaffected in this phase.

3.1.3 Intermediate Fields

After field splitting, the type becomes a bit more simple, with a list of *intermediate fields* A string is either a quoted string, `"s"` or an unquoted string, `s` and we track field separators created from non-whitespace characters, $\underline{\texttt{c}}$, separately from whitespace field separators, ␣ . Pathname expansion will step through our list of intermediate fields and for any unquoted strings, attempt to expand the any regular expression patterns to filenames.

As the final step of *word expansion*, quote removal, needs to produce *fields* as a result. This means that both field separator types and the `"s"` types must be removed. For a quoted string, as the name implies, the quotes are removed and it is treated as a regular string.

3.2 Symbolic

The symbolic type, λ, shows up throughout the types above. A symbolic word could be anything, since we have not implemented command evaluation we cannot know the output of a command. The symbolic type allows us to continue to make progress on the parts that are not commands. As we complete the model for expansion, fewer symbolic types will be generated. Currently utilities such as `echo`, `mv`, `rm` all have symbolic results but their operation is specified by the POSIX specification so in a complete model of the shell these utilities must be modeled. Even so, we will always need a symbolic type. Not all commands have been specified by the POSIX shell, in fact most have not. By including a symbolic type, we can correctly handle, but ignore the results of command substitution when the command evaluated is not specified by POSIX.

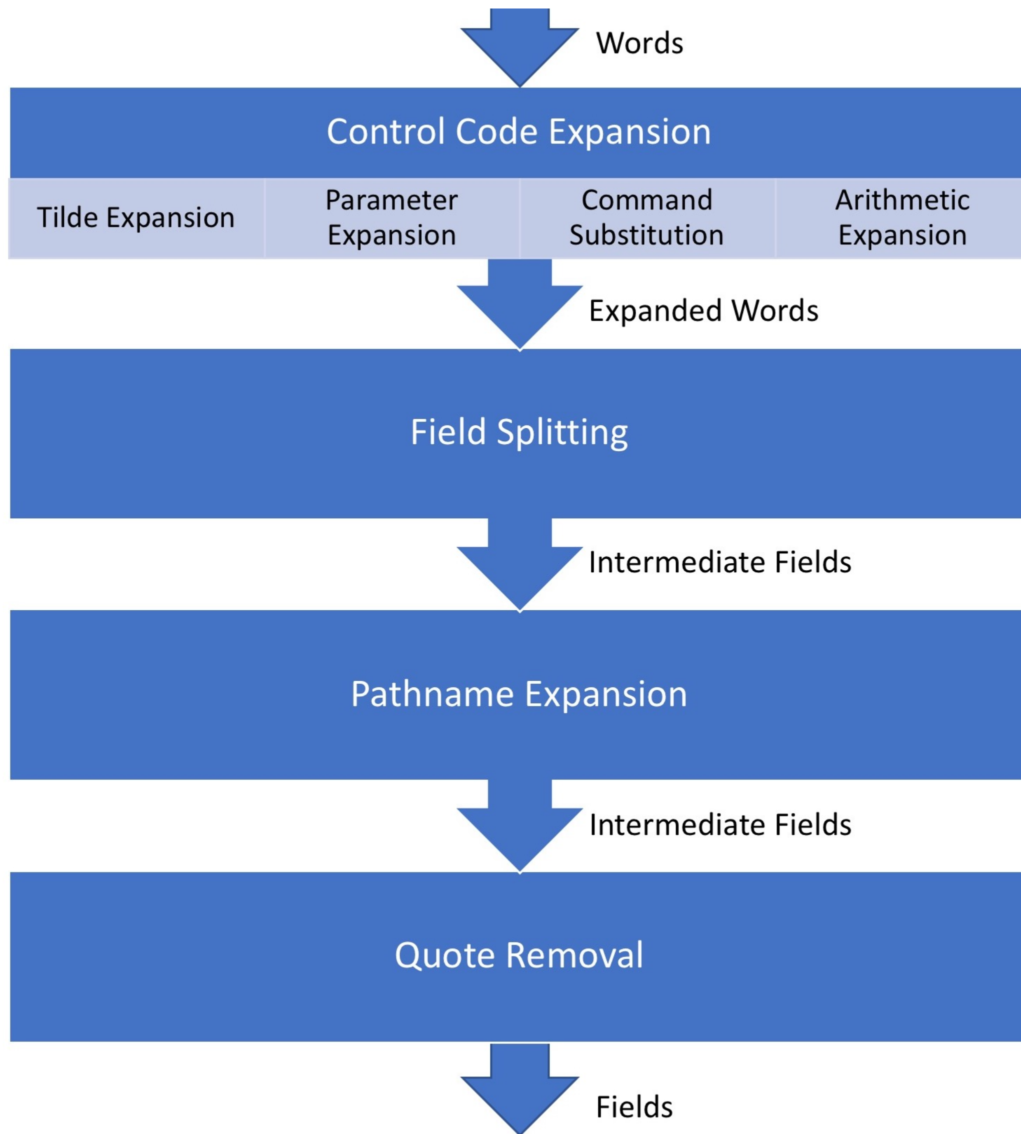


Figure 3.1: Operation Semantics Flowchart

Syntax

Strings	$\mathsf{s} \in \Sigma^*$	where Σ is, e.g., UTF-8
Parameter formats	$\phi \in \Phi$	$::= \bullet \mid \texttt{-}w \mid \texttt{:-}w \mid \texttt{=}w \mid \texttt{:=}w$
		$\mid \texttt{?}w \mid \texttt{:?}w \mid \texttt{+}w \mid \texttt{:+}w \mid \texttt{\#}$
		$\mid \texttt{\%}w \mid \texttt{\%\%}w \mid \texttt{\#}w \mid \texttt{\#\#}w$
Control codes	$k \in \mathsf{K}$	$::= \texttt{\~{}} \mid \texttt{\~{}s} \mid \texttt{\$\{s} \mid \phi\} \mid \texttt{\$(}ws\texttt{)} \mid \texttt{\$((}ws\texttt{))} \mid \texttt{"}ws\texttt{"}$
		$\mid \texttt{\$\{s}\ assign\ \ expws\,\texttt{;}\,ws\}$
		$\mid \texttt{\$\{s}\ match\ \ \mathbf{ss}\ \mathbf{sm}\ \ expws\,\texttt{;}\,ws\}$
		$\mid \texttt{\$\{s}\ error\ \ expws\,\texttt{;}\,ws\}$
Fields	$f \in \mathsf{F}$	$::= [] \mid \texttt{s};f$
Word	$w \in \mathsf{W}$	$::= \texttt{s} \mid k \mid ␣ \mid \lambda$
Words	$ws \in \mathsf{WS}$	$::= [] \mid w; ws$
Expanded Word	$expw \in \mathsf{EXPW}$	$::= ␣ \mid \texttt{exp s} \mid \texttt{s} \mid \texttt{"s"} \mid \lambda$
Expanded Words	$expws \in \mathsf{EXPWS}$	$::= [] \mid \mathbf{expw}; expws$
Intermediate Field	$intf \in \mathsf{INTF}$	$::= ␣ \mid \underline{\texttt{c}} \mid \texttt{s} \mid \texttt{"s"}$
Intermediate Fields	$intfs \in \mathsf{INTFS}$	$::= [] \mid intf; intfs$
Commands		$::= \lambda$
Patterns	$ps \in \mathsf{P}$	$::= w \mid w\vert ps$
System state	$sys \in \mathsf{Sys}$	$= \langle \mathsf{fs}, \mathsf{P}, \rho \rangle$
Filesystem state	$\mathsf{fs} \in \mathsf{FS}$	
Process state	$\mathsf{P} \in \mathsf{Proc}$	
Shell state	$\rho \in \mathsf{Env}$	

Figure 3.2: Syntax

Chapter 4

Mechanized Semantics

The mechanized semantics for POSIX word expansion operate by stepping through the expansion process and recording a trace of system and program state. The program state is recorded as the various types described in Chapter 3. For example, the record of control code expansion will be recorded by multiple entries of Expanded Words and Words as Words are gradually expanded in to Expanded words. The field splitting step is recorded as a step from Expanded Words to Intermediate Fields. Pathname expansion is a step that may change the values of intermediate fields, but not the types, and finally quote removal will finalize the type of fields.

Since we are only interested in modeling the expansion and evaluation aspects of POSIX, the input to our mechanized semantics is received directly from the Dash parser. An OCAML shim connects this parser to our executable model. This means that certain things understood as features of the POSIX shell are not explicitly modelled by our semantics. As we covered in Chapter 2, space and tab characters are *always* interpretted as field separators, regardless of `IFS` value, when they are entered by the user. If you analyze our mechanized semantics you won't see that feature described anywhere because it is handled during parsing. Instead, you'd have to notice that the input to our model is, Words, and a field separator is a Word. A field separator is also an Expanded Word and is stilled tracked in an intermediate field. Only when we construct the final Fields result do we drop the field separator type. This is because at the very end, we no longer need to track field separators for evaluation, so our model is a list of strings, all of which were separated by field separators in the Intermediate Fields representation.

Each trace entry in generated by the `step_expansion` function. This

function takes as its input the current state of expansion, and based upon the information given, performs a single step of expansion, returning the result. The trace is generated by successive calls to this function and the recording of each output. Each output is tagged with the step of expansion that generated it which aids in being displayed by the visualization tool.

The `step_expansion` will initally receive words, which is will immediately begin expanding. All four control code expansions *must* happen at the same time because changes to the system state in one control code expansion can affect the next (e.g. `echo $x=5 $((x+2))` will expand to `echo 5 7` if the assignment to `x` occurs). If we did not process control code expansions in left to right order, we could not properly handle this feature. This, and the fact that control code expansion can result in nested expansion, means that many of the steps taken by the model will be as a result of expanding control codes. This differes from implementations of the POSIX shell, which perform all seven expansions in a single left to right pass. This means that each Word input goes through each of the four expansion states (control code expansion, field splitting, pathname expansion, and quote removal. By using separate types at different phases, the model can process the input in multiple left to right passes and generate a trace of expansions that shows all field splitting, all pathname expansions, and all quote removal steps in one step each.

While our model did not have to implement the POSIX parser, it did need to implement an arithmetic parser. Arithmetic expansion uses a runtime parser-evaluator to turn strings into arithmetic expressions. Since our variables are stored as strings, that means the same expansion method can be used to expand `$x` and `$(($x+2))`. The nested `$x` is first expanded and the string `5+2` is given to the arithmetic parser. You'll notice in other examples, I omitted the `$` inside the nested parameter expansion. Both styles are specified by the POSIX shell, but they will produce slightly different traces in our model.

Our step semantics treat arithmetic parsing and evaluation as a black box and does not record the steps taken. The result is that `$(($x+2))` will produce trace steps that `$((x+2))` will not. This is because prior to arithmetic parsing, the string contained undergoes control code expansion. This means that our model will record a trace of the parameter expansion `$x`. But the parser does not identify `x` as a control code, and so leaves the string unchanged. When the arithmetic parser-evaluator attempts to evaluate this string it will realize it cannot parse it as a number and attempt to look up its value then. Since this parameter lookup occurs inside the "black box" of arithmetic parsing and evaluation, no trace of it will be recorded.

Pathname expansion (and pattern matching inside parameter expansion) required the implementation of a regular expression parser, which suffers similar limitations in the trace provided as arithmetic expansion. Much like the contents of arithmetic expansion, the contents of a pattern match will undergo control code expansion. For example, `x=foo; $x*` will expand `$x` to `foo` before attempting to match with the pattern `foo*`. Since this was a control code expansion its steps will be traced. The process of matching a pattern to a string is the "black box". This regular expression matcher does not track its steps, it only produces a result in the form of "matched" or "did not match".

Quote removal is simple because the model has tracked quoting with special types throughout the process. The real work of modelling quotes happens in tracking them appropriately throughout the process and preventing field splitting from occuring when it should not. Quotes will also prevent certain other special characters from expanding. For example, `"~"` is not expanded in tilde expansion and `"*"` does not undergo pathname expansion. Both of these cases are handled by the parser because the tilde character must be the first character in its word (but in the above case, the quotation mark is) and patterns must be unquoted, but due to the mixed definitions of parsing and evaluation in the POSIX specification, care should be taken in ensuring the Dash parser behaves as we would expect it to in cases such as these.

Chapter 5

Related Work

5.1 Background

The development of a programming language most commonly begins with an implementation of that language. The development of the C programming language began in 1969, and by 1972 a sizable part of the Unix operating system was written in C. Over the coming 18 years, C would be implemented on many different systems with no standard to follow. This created an ecosystem where the type of C you were programming in was dependent on which implementation your system had on it. It was not until 1989, 20 years after the development of C began, that the first standard specification would be published. This became known as Standard C, C89, or C90 (all of which refer to the same language) [Rit93].

The C language is not the only language with this developmental trajectory, the early stages of graphical web browsers had significant conflicts due to differences in JavaScript implementations. JavaScript began as an implementation inside of Netscape's Mosaic browser (in 1995) in order to give web developers the ability to create more dynamic web pages. In order to keep up with the competition, Microsoft soon included a version of JavaScript in Internet Explorer 3 that was based on a reverse engineering of Netscape's work. This inevitably lead to major differences in the two browsers. Most websites found it too expensive to develop a page compatible with both browsers' JavaScript implementation so they resorted to applying "*best viewed in Netscape*" or "*best viewed in Internet Explorer*" labels to their sites. This lead to the immediate need for standardization, which was provided by the standard specification of ECMAScript in 1997.

These standard specifications are usually written as a combination of English and psuedocode and as such are verbose and difficult to read. In or-

der to better understand a specification, programming language researchers aim to develop a formal model of the specification. A formal model consists of mathematical definitions for how a program's Abstract Syntax Tree (AST) executes. Formal models can take many forms, some are designed to help create formal proofs of how a program executes, some are designed to produce an "oracle" against which compiler writers can verify their work. These goals then influence the type of formal model that is designed.

5.1.1 Formal Models

A formal model can take many forms, as Ellison & Rosu describe in their brief overview of previous work on the C language, for the first attempt at a formal model of C Gurevich & Huggins used Abstract State Machines, other work used a custom-made temporal logic, Norrish defined a formal semantics using the HOL theorem prover [ER12, Nor98]. The downside to these were that they did not produce an executable version of this model. For the C language this paper will mainly focus on the memory models and operational semantics of Ellison & Rosu and Krebbers both of which are written in Coq, which can be translated into an executable Ocaml program [ER12, KW15]. For JavaScript, Gardner [BCF+14] creates a Coq specification (JSCert) that is extracted to an Ocaml executable (JSRef). The formal model created for the POSIX shell that is described in this paper is written in Lem [OBZNS11], a language that can be compiled to Coq (for proofs) and Ocaml (for execution).

5.1.2 Coq

By formalizing results in the Coq proof assistant, the designers of formal models can have confidence that their semantics are mathematically well-formed, but it does not guarantee that the program semantics defined are the correct. Constructing the semantics requires careful reading of the specification. One of the main advantages of creating in executable from the formal semantics is to provide further confidence in the correctness of the semantics by running tests against the executable semantics. During development of a formal model, it will be beneficial to run unit tests to ensure the pieces of model are functioning as specified. Once the model is advanced enough, they can be tested (in combination with a parser) against a compiler's test suite. Individual compilers have test suites to ensure that the many implementations across different systems all behave in a similar manner. These test suites provide large, and often complicated, tests that

c

$$\frac{e_1 \text{ eval } n_1 \quad e_2 \text{ eval } n_2 \quad n = n_1 + n_2}{e_1 \ + e_2 \longrightarrow \ n} \text{ (SIMPLEPLUS)}$$

a formal semantics can be run against. For C, the obvious choice would be the GCC test suite, Ellison & Rosu [ER12] test against what is called the "GCC torture test suite", which is a subset of the GCC test suite that contains many of the edge-cases that GCC compilers have missed over the years.

5.2 Defining the Operational Semantics

Every formal model needs to define the semantic rules. These rules describe how the language's AST will execute by defining step evaluation rules. In order to get to the AST, formal models will frequently make use of an existing compiler's parser. Ellison & Rosu [ER12] and Krebbers [KW15] both make use of the CIL parser for C, Gardner [BCF+14] relies on a JavaScript parser taken from the Google Closure Compiler, and our mechanized semantics for the Shell uses the parser for Dash, a POSIX-compliant shell that is the standard command interpreter for Linux.

After obtaining the AST from the parser, a formal model is not required to stay limited to the structure of the tree. In order to simplify the model, it is often helpful to use one or more intermediate representations of the AST with more information available to the model. Ellison and Rosu choose to operate directly on the C language AST [ER12]. Similarly Bodin and Gardner operate directly on the JavaScript AST [BCF+14]. Whereas Krebber uses both an intermediate abstract syntax that is created from the result of the CIL parser and a core syntax, which is a simplification of the custom abstract syntax that ensures a weak type-safety [KW15].

Now that we have the AST (potentially in a modified form) we can begin to define evaluation rules for the various types of AST elements. Defining evaluation rules can seem straight-forward in the beginning, but it frequently proves quite complicated. For example, lets take a look at the standard definition for addition as a semantic step rule presented by Greenberg [Gre17]. The SIMPLEPLUS rule defined below says that if e_1 evaluates to a number, n_1, and e_2 evaluates to a number, n_2, and the sum of those numbers is n then $e_1 + e_2$ steps to n.

It's tempting to think that a rule such as this is sufficient for every arithmetic operation. Unfortunately, as we'll see in later sections, even something as simple as addition requires multiple evaluation rules in C due to the potential for signed integer overflow and other non-standard addition behaviors. In order to design a formal model for *actual* addition in a programming language, Ellison & Rosu [ER12] describe a system of defining small-step rules for evaluating expressions, and combining them into larger big-step semantics for evaluating statements such as the rule described above.

5.2.1 Challenges of Underspecification

The main challenge in defining an operational semantics arises from underspecification. Specifications often deliberately (and occasionally on accident) leave certain details of evaluation to be defined by the compiler writers. This gives compiler writers the flexibility to do what is best for their system and to implement optimizations, but can cause problems for programmers. A program can behave properly on one system but not on another because the program relied on implementation-defined or unspecified behavior. These underspecifications normally fall into three categories: implementation-defined, unspecified, and undefined behavior.

Implementation-defined Behavior

Implementation-defined behavior is used to allow compiler designers to optimize their implementation for the given architecture. For example the number of bytes used to represent an integer is not defined by the C standard, and it is instead left to a compiler writer to decide on a number of bytes [Kre15]. This information is important to the execution of a program, so the C standard requires that the choices for implementation-defined behavior be documented.

In C, multiplication provides a good foundation from which to investigate the consequences of implementation-defined behavior. First, it is important to know that when multiplying signed integers multiplicative overflow is undefined, an error. The code below uses signed integers and performs a simple squaring operation, so how will it operate?

```
int x = 65535;
int y = x * x;
```

If an integer uses 8 bytes (64 bits) to store the value of an |int— the value of |y— will be $4,294,836,225$, which is also the mathematical result of that

multiplication. Following the C standard, if an implementation used only 4 bytes (32 bits) a |y— value of $-131,071$ is an equally correct C program, as is any other number, and even an error. By making the result of signed multiplication undefined, the C standard intends to support multiplication by repeated addition or by a specialized multiplcation circuit in the CPU [ER12].

A solution to modelling implementation-defined behavior is described by Krebbers [Kre15] as "parameterization". You can think of this like Java interfaces, the `List` interface defines what it means to be a list, just like a formal semantics define what it means to evaluate a program in that language, but some of the details of a `List` are left to the implementing class such as `ArrayList` and `LinkedList`, which will determine the details of how the `List` functions. A more accurate analogy would be to this is like a type-class in Haskell. If something implements the `Num` type-class, it implements a set of functions (in the case of `Num`, a minimal complete definition requires the following functions to be implemented `(+), (*), abs, signum, fromInteger, negate`) that allow it to function like a `Num`. We can define our semantics to perform arithmetic on a `Num` and then implement types such as `int32` for 32-bit integers and `int64` for 64-bit integer that implement the `Num` type-class, but operate as if they were an integer of their respective size.

In the case of integers Krebber's $\mathtt{CH_2O}$ semantics are parameterized by the size and whether or not the integers are represented using big- or little-endian. The POSIX shell specification defers to the C specification of signed long integers for its arithmetic and so would be subject to similar parameterization [IG18]. The POSIX shell also has a wide variety of other implementation-defined behavior. The default `PS1` value and the largest integer that can be used as a file descriptor in a redirect (the number 1 in `echo foo 1> /dev/null` is a file descriptor) are implementation-defined. However, a formal model of JavaScript benefits from not needing to parameterize arithmetic in a similar fashion. The ECMAScript specification specifies that all integers are represented with their Number class, which uses a 64-bit floating point number [Int17].

A formal model of JavaScript does present some other interesting challenges with implementation-defined behavior. A function is allowed to be called with an arbitrary number of arguments with the extras being stored in an `args` array. The ECMAScript specification thus allows many of the core library and object functions to process additional arugments in any implementation-defined way. The default way is to ignore extra arguments because most functions never interact with the `args` array. Some more complicated implementation-defined behavior can be found in

`Number.prototype.toExponential(fractionDigits)`, which will return a string of the number in exponential form with `fractionDigits` decimal places. The ECMAScript specification allows the function to throw a `RangeError` when `toInteger(fractionDigits) > 20`, but it is also allowed to extend the values of `fractionDigits` it accepts. Some initial testing in Firefox Version 57.0 (64-bit) shows that it does extend the functionality of `toExponential` to accept any input for `fractionDigits` such that `toInteger(fractionDigits) <= 100`. It prints out correct output up to 53 decimal places and inserts 0 for any decimal places after 53. As a result, a full formal model of JavaScript would need to be parameterized by **every** decision made for how to handle additional arguments and to what extent it relaxes the requirements of various function parameters, such as the example described above [Int17]. These challenges are significant and Gardner [BCF+14] choose to simplify their model by making "arbitrary but natural choices" for implementation-defined behavior and following ECMAScript specifications strictly. The natural choice for `toExponential` would be to throw a `RangeError` if the input violated the `toInteger(fractionDigits) > 20` requirement discussed above. This also avoid the situation where the number of decimal places exceeds 53, the maximum number of decimal places representable by a 64-bit double.

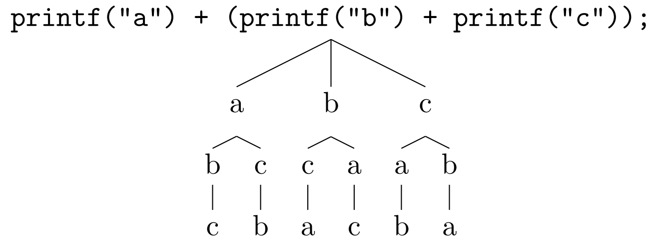
Unspecified Behavior

Unspecified behavior also provides an opportunity for optimization. The C specification allows the evaluation of an expression in any order [Kre15]. This is most easily seen in the evaluation of arithmetic expressions. The compiler is free to evaluate the expression in whichever order it thinks will perform the best, hopefully taking advantage of an architecture feature such as pipelining. The order of evaluation in a single C statement is very loosely defined. Take a look at this example given by Krebbers [KW15].

```
printf("a") + (printf("b") + printf("c"));
```

Looking at the code above and knowing that addition may be evaluated in either order one may assume that the valid outputs are `abc` or `bca`. But there are even fewer restrictions on the C compiler. It is allowed to execute both sides of each plus operation in either order and even interleave the evaluations of each piece (it can be evaluating parts of each `printf` function at the same time). Since the restriction on evaluation order are so relaxed, `acb` and even `cba` are also correct outputs. A complete formal model of C needs to account for all the valid execution paths of an expression.

From a high level view, the formal model would need to model an execution tree something like the execution tree below, where each path represents a possible printed order.



Unspecified behavior can be handled by non-deterministic small-step semantics. A non-deterministic step in a semantic rule will create a branch in the execution tree where each path from root to leaf node represents a possible evaluation path. Therefore for the C program print example shown above, a correct formal model of the C standard would not merely say that the output is `abc` or another permutation, but it would give all the valid permutations that still comply with the C standard. Thus, non-deterministic evaluation rules for addition in C might look something like the following set of rules. Keep in mind, these are still not sufficient to fully model addition in C because they do not handle overflow.

$$\frac{e_1 \ \mathsf{step}\ e_1'}{e_1 \ + e_2 \longrightarrow \ e_1' + e_2} \ (\textsc{StepLeft}) \qquad \frac{e_2 \ \mathsf{step}\ e_2'}{e_1 \ + e_2 \longrightarrow \ e_1 + e_2'} \ (\textsc{StepRight})$$

$$\frac{e_1 \ \mathsf{step}\ n_1}{e_1 \ + e_2 \longrightarrow \ n_1 + e_2} \ (\textsc{StepLeftValue}) \qquad \frac{e_2 \ \mathsf{step}\ n_2}{e_1 \ + e_2 \longrightarrow \ e_1 + n_2} \ (\textsc{StepRightValue})$$

$$\frac{n_1 + n_2 = n}{n_1 \ + n_2 \longrightarrow \ n} \ (\textsc{StepValue})$$

In the above rules, StepLeft and StepRight represent the non-deterministic small-step evaluation. Returning to the `printf` example, if you assume the first call of `step` on `printf` prints the argument and the second reduces it to a number, any of the print orders can be obtained via careful application of the above two rules. Then StepLeftValue and StepRightValue would

evaluate the `printf` function to its integer return value, and the StepValue rule would reduce the three values to a single result.

Undefined Behavior

Undefined behavior arises for program semantics that the standard declares illegal. Specifications frequently impose no requirements for the program in these situations. The program is allowed to crash or perform any arbitrary behavior. In C, undefined behavior is easy to create by dereferencing a `NULL` or dangling pointer, causing signed integer overflow, or the more subtle sequence pointer violation [Kre15]. In the POSIX specification, if a standard utility specifies that it takes a "text file" as an argument and is given a non-text file (like a directory) the behavior is undefined [IG18].

Lets take a look at a more interesting case of undefined behavior in C than multiplicative overflow. In order to allow the unspecified execution order of a single line of C code, the standard requires that all execution orders satisfy the sequence point restriction, which "does not allow an object to be modified more than once (or being read after being modified) between two sequence points" [KW15]. For simplicity you can think of a sequence point as a semi-colon, but conditionals and function calls also have their own notion of a sequence point.

```
1    int x;
2    int y = (x = 3) + (x = 4);
3    printf("%d %d\n", x, y);
```

Considering all the execution orders, this program could print `3 7` or `4 7` depending on the execution order of the addition on line 2. Krebbers [KW15] say that GCC (version 4.6.4 with the `-O2` flag) correctly prints `4 8`, which does not represent an allowed execution order. No implementation-defined or unspecified behavior could account for the number 8 being assigned to y, but it turns out the above example exhibits undefined behavior. Outputing `4 8` is correct relative to the C standard because line 2 is causing a sequence point violation by assigning to x twice without encountering a sequence point. This means that the behavior of line 2 is undefined, so the behavior of the program as a whole is undefined. Since this is undefined behavior, the program could have done anything, even errored and crashed, and still followed the C standard.

Undefined behavior presents a challenge to designers of formal models, the behavior is not parameterized by any logical requirements nor is the

program required to continue executing. An undefined behavior is potentially an unrecoverable error, but every implementation can choose their own way to handle it. Krebbers [Kre15] describes the solution as an "absence of semantics". This means that in their model whenever a C program encounters an undefined behavior it enters an `undef` state and ends. If you have ever encountered the error `Segmentation fault: 11` this is equally unsatisfying. But entering an `undef` state is *actually* the best a model can do. Some implementations might error, some might recover, some might recover or error for the same semantic based off of some other information in the program. When compiler designers can do *anything* it is impossible to model and the absolute best a model can do is give up.

5.3 Modelling Memory

The most literal model of memory for any program would be an array of bytes. Regardless of the system, all computers store data as an array of bytes where the memory address of that data is the index. Then the model would need to define the semantics by which those bytes can be used to represent the various features of the langauge. While executable code is restricted to storing its information in an array of bytes, memory models used in formal verification are not, and should not, be bound by that restriction. As the end goal of a formal model is to create a mathematical representation of a programming langauge, formal models are allow to, and most often should, use more complex data types to aid in modelling the program memory.

Ellison & Rosu [ER12] choose a fairly similar memory model with a map from ID's (memory addresses) to arrays of bytes representing the data. For C, this is a "good enough" memory model. They are able to define a formal semantics that allows them to pass most of the GCC torture test suite. Krebbers [KW15] show that a more complicated memory model allows their model to reason about a new feature a "well-typed" C program. Their memory model consists of a map from ID's (pointers) to tree structures, these tree structures correspond to the structure of the data being represented. Think about a `struct` with 4 integer fields. Instances of this struct would then have 4 leaf nodes of the parent. In their model, the leaf nodes are the bytes used to actually store the data. By using a more informed memory model, Krebbers is able to reason about features of a C program that Ellison & Rosu are not.

JavaScript, as a higher-order language, presents a more complicated memory model that is actually defined in the specification. Pulled directly

from the ECMAScript definition, "the memory model defines both the precise conditions under which a program exhibits sequentially consistent behaviour as well as the possible values read from data races. To wit, there is no undefined behaviour" [Int17]. This is a bold claim, but a boon for a formal model of JavaScript. If faithfully implemented, it would be possible for a memory model to always produce a result.

5.4 Comparison

The formal models of the three languages (C, JavaScript, and POSIX) are extremely useful in understanding each language. The specifications for these languages are verbose and difficult to process, but having formal models of each language allows researchers and developers to investigate the semantics of each language in a complete way (assuming the model is complete). This paper has included a number of seemingly simple programs that can behave widely differently based on implementation-defined and unspecified behaviors.

Even though the languages differ greatly, the approach to building a formal model for each shares many similarities. This is because the main challenge is underspecification. If specifications were entirely specified there'd be no need for a formal model, but then languages would be much less portable and much more difficult to optimize. Language specific features merely dictate the details of each step to creating a formal model and will change which steps are the most difficult, but each language shares a general road map for constructing a formal model.

Chapter 6

Future Work and Conclusion

6.1 Future Work

The next obvious step is to complete the mechanized semantics for command evaluation so that we have a complete model of the Shell. This involved handling a portion of the POSIX file system as well. Once this model is complete, what we have is another shell. We could swap out the symbolic file system calls for the actual file system calls and our executable semantics would function identically to an existing shell.

Once we have a full executable model of the shell, it could be tested for POSIX-compliance. If our model if fully POSIX-compliant, we know we have correct operational semantics (even if they might not be the absolute simplest). Having an executable model of our semantics is helpful because it allows us to use existing test suites (like those for the POSIX shell) to verify the rules we have chosen.

Since we can compile out operational semantics to Coq, we can also use our semantics in formal proofs. We might be interested in proving that certain simplifications (optimizations) don't effect the outcome, or that adding new features won't adversely affect old features. For example, as we've seen throughout this paper, it would be wonderful to be able to specify that a variable should not expand to more than one field at assignment. This is a feature that could be implemented in a backwards compatible manner, and Coq could help us prove that.

6.2 Conclusion

The mechanized semantics for word expansion in the Shell is a good first step towards creating a full model of Shell execution. The model we have provided provides a clear trace of how Shell expansion happens, and what effects it has on the shell environment. And while the existing trace provides a nice visual way to investigate shell expansion, future work will enable even more robust verification of Shell execution.

Bibliography

[BCF+14] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised javasript specification. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 87–100. ACM, 2014.

[Deb] Debian. https://wiki.debian.org/shell.

[ER12] Chucky Ellison and Grigore Rosu. An executable formal semantics of c with applications. In *ACM SIGPLAN Notices*, volume 47, pages 533–544. ACM, 2012.

[Gre17] Michael Greenberg. Understanding the posix shell as a programming language. *Off the Beaten Track*, 2017.

[IG18] IEEE and The Open Group. The open group base specifications issue 7 (ieee std 1003.1-2017). 2018.

[Int17] Ecma International. Standard ecma-262: Ecmascript language specification. 2017.

[Kre15] Robbert Jan Krebbers. *The C standard formalized in Coq*. [Sl: sn], 2015.

[KW15] Robbert Krebbers and Freek Wiedijk. A typed c11 semantics for interactive theorem proving. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 15–27. ACM, 2015.

[Nor98] Michael Norrish. C formalised in hol. Technical report, University of Cambridge, Computer Laboratory, 1998.

[OBZNS11] Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell. *Lem: A Lightweight Tool for Heavyweight Semantics*, pages 363–369. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[Rit93] Dennis M Ritchie. The development of the c language. *ACM SIGPLAN Notices*, 28(3):201–208, 1993.

[RST+15] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. Sibylfs: Formal specification and oracle-based testing for posix and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 38–53, New York, NY, USA, 2015. ACM.