A Formally Verified Interpreter for a Shell-like Programming Language[⋆]

Nicolas Jeannerod[1,2], Claude Marché[3], and Ralf Treinen[2]

[1] Dpt. d'Informatique, École normale supérieure, Paris, France
[2] Univ. Paris Diderot, Sorbonne Paris Cité, IRIF, UMR 8243, CNRS, Paris, France
[3] Inria & LRI, CNRS, Univ. Paris-Sud, Université Paris-Saclay, Orsay, France

Abstract. The shell language is widely used for various system administration tasks on UNIX machines, as for instance as part of the installation process of software packages in FOSS distributions. Our mid-term goal is to analyze these scripts as part of an ongoing effort to use formal methods for the quality assurance of software distributions, to prove their correctness, or to pinpoint bugs. However, the syntax and semantics of POSIX shell are particularly treacherous.
We propose a new language called CoLiS which, on the one hand, has well-defined static semantics and avoids some of the pitfalls of the shell, and, on the other hand, is close enough to the shell to be the target of an automated translation of the scripts in our corpus. The language has been designed so that it will be possible to compile automatically a large number of shell scripts into the CoLiS language.
We formally define its syntax and semantics in Why3, define an interpreter for the language in the WhyML programming language, and present an automated proof in the Why3 proof environment of soundness and completeness of our interpreter with respect to the formal semantics.

1 Introduction

The UNIX shell is a command interpreter, originally named *Thompson shell* in 1971 for the first version of UNIX. Today, there exist many different versions of the shell language and different interpreters with varying functionalities. The most popular shell interpreter today is probably the *Bourne-Again shell* (*a.k.a.* bash) which was written by Brian Fox in 1988 for the GNU project, and which adds many features both for batch usage as an interpreter of shell scripts, and for interactive usage.

We are interested in a corpus of *maintainer scripts* which are part of the software packages distributed by the Debian project. The shell features which may be used by these scripts are described in the *Debian Policy [18], section 10.4, Scripts*. Essentially, this is the shell described by the POSIX [12] standard. In the rest of the paper we will just speak of "shell" when we mean the shell language as defined by the POSIX standard.

[⋆] This work has been partially supported by the ANR project CoLiS, contract number ANR-15-CE25-0001.

Maintainer scripts are run as the *root* user, that is with maximal privileges, when installing, removing or upgrading packages. A single mistake in a script may hence have disastrous consequences. The work described in this paper is part of a research project with the goal of using formal methods to analyse the maintainer scripts, that is to either formally prove properties of scripts as required by the Debian policy, or to detect bugs. The corpus contains, even when ignoring the small number of scripts written in other languages than POSIX shell, more than 30.000 scripts.

Verifying shell scripts is a hard problem in the general case. However, we think that the restriction to Debian maintainer scripts makes the problem more manageable, since all the scripts are part of the common framework of the Debian package installation process, and the Debian policy tells us how they are called, and what they are allowed to do. For instance, the package installation process is orchestrated by the `dpkg` tool which guarantees that packages are not installed in parallel, which justifies our decision to completely ignore concurrency issues. The installation scripts are indeed often simple and repetitive. They are written by package developers, who have, in general, good knowledge of the shell; they try to avoid bad practices, and are quite aware of the importance of writing modular and maintainable code.

Even in that setting, the syntax and the semantics of shell is the first obstacle that we encounter during our project, since they can be treacherous for both the developer and the analysis tools. We have written a parser and a statistical analyser for the corpus of shell scripts [14] which we used in order to know which features of the shell are mostly used in our corpus, and which features we may safely ignore. Based on this, we developed an intermediate language for shell scripts, called *CoLiS*, which we will briefly define in this paper. The design of the CoLiS language has been guided by the following principles:

– It must be "cleaner" than shell: we ignore the dangerous structures (like `eval` allowing to execute arbitrary code given as a string) and we make more explicit the dangerous constructions that we cannot eliminate.
– It must have clear syntax and semantics. The goal is to help the analysis tools in their work and to allow a reader to be easily convinced of the soundness of these tools without having to care about the traps of the syntax or the semantics of the underlying language.
– The semantics must be less dynamic than that of the shell. This can be achieved by a better typing discipline with, for instance, the obligation of declaring the variables and functions in a header.
– An automated translation from shell must be possible. Since the correctness of the translation from shell to CoLiS cannot be proven, as we will argue in the following, one will have to trust it by reading or testing it. For this reason, the CoLiS language cannot be *fundamentally* different from shell.

This language is not conceived as a replacement of shell in the software packages. If that was our goal, we would have designed a declarative language as a replacement (similar to how systemd has nowadays mostly replaced System-V init scripts). Our mid-term goal is to analyse and, in the end, help to improve the existing shell scripts and not to change the complete packaging system. Because of this, our language shares a lot of similarities (and drawbacks) with shell.

We have formally defined the syntax and semantics of CoLiS in the Why3 verification environment [5]. It is already at this stage clear that we will be faced with an important problem when we will later write the compiler from shell to CoLiS: how can we ensure the correctness of such a compiler? The root of the problem is that there simply is no formal syntax and semantics of the shell language, even though there are recent attempts to that (see Section 5). In fact, if we could have clean syntax and semantics for the shell, then we wouldn't need our intermediate language, nor this translation, in the first place. An *interpreter* plays an important role when we want to gain confidence in the correctness of such a compiler, since it will allow us to compare the execution of shell scripts by real shell interpreters, with the execution of their compilation into CoLiS by the CoLiS interpreter. The main contribution of this paper is the proof of correctness and completeness of our CoLiS interpreter, with respect to the formal semantics of CoLiS.

Plan of the paper. We present the syntax and semantics of our language in Section 2. We also explain some of our design choices. We describe our interpreter in Section 3 and the proof of its completeness in Section 4. This proof uses a technique that we believe to be interesting and reusable. Finally, we compare our work to other's in Section 5 and conclude in Section 6.

2 Language

2.1 Elements of Shell

Some features of the shell language are well known from imperative programming languages, like variable assignments, conditional branching, loops (both `for` and `while`). Shell scripts may call UNIX commands which in particular may operate on the file system, but these commands are not part of the shell language itself, and not in the scope of the present work. Without going into the details of the shell language, there are some peculiarities which are of importance for the design of the CoLiS language:

Expressions containing instructions. Expressions that calculate values may contain control structures, for instance a `for` loop, or the invocation of an external command. Execution of these instructions may of course fail, and produce exceptions.

No static typing. Variables are not declared, and there is no static type discipline. In principle, values are just strings, but it is common practice in shell scripts to abuse these strings as lists of strings, by assuming that the elements of a list a separated by the so-called *internal field separator* (usually the blank symbol).

Dynamic scoping. Functions may access non-local variables, however, this is done according to the chronological order of the variables on the execution stack (dynamic scoping), not according to the syntactic order in the script text (lexical scoping).

String variables	x_s	$\in SVar$
List variables	x_l	$\in LVar$
Procedures names	c	$\in \mathcal{F}$
Natural numbers	n	$\in \mathbb{N}$
Strings	σ	$\in String$
Programs	p	$::= vdecl^* \ pdecl^* \ \textbf{program}\ t$
Variables declarations	$vdecl$	$::= \textbf{varstring}\ x_s \mid \textbf{varlist}\ x_l$
Procedures declarations	$pdecl$	$::= \textbf{proc}\ c\ \textbf{is}\ t$
String expressions	s	$::= \textbf{nil}_s \mid f_s :: s$
String fragments	f_s	$::= \sigma \mid x_s \mid n \mid t$
List expressions	l	$::= \textbf{nil}_l \mid f_l :: l$
List fragments	f_l	$::= [s] \mid \textbf{split}\ s \mid x_l$
Terms	t	$::= \textbf{true} \mid \textbf{false} \mid \textbf{fatal}$
		$\mid \textbf{return}\ t \mid \textbf{exit}\ t$
		$\mid x_s := s \mid x_l := l$
		$\mid t\ ;\ t \mid \textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t$
		$\mid \textbf{for}\ x_s\ \textbf{in}\ l\ \textbf{do}\ t \mid \textbf{do}\ t\ \textbf{while}\ t$
		$\mid \textbf{process}\ t \mid \textbf{pipe}\ t\ \textbf{into}\ t$
		$\mid \textbf{call}\ l \mid \textbf{shift}$

Fig. 1. Syntax of CoLiS

Non-standard control flow. Some instructions of the shell language may signal exceptional exit, like a non-zero error-code. Different constructions of the shell language propagate or capture these exceptions in different ways. This has sometimes quite surprising consequences. For instance, `false && true` and `false` are not equivalent in shell. Furthermore, there is a special mode of the shell (the *strict* mode, in Debian parlance, obtained by the `-e` flag), which changes the way how exceptions are propagated.

2.2 Syntax of CoLiS

The shell features identified in Section 2.1 motivate the design of the CoLiS language, the syntax of which is shown in Figure 1. There only is an abstract syntax because the language is meant to be the target of a compilation process from the shell language, and is not designed to be used directly by human developers.

Terms and Expressions The mutual dependency between the categories of instructions and expressions which we have observed in the shell does not pose any real problem, and shows up in the definition of the CoLiS syntax as a mutual recursion between the syntactic categories of terms (corresponding to instructions), expression fragments, and expressions.

Variables and typing All the variables must be declared. These declarations can only be placed at the beginning of the program. They are accompanied by a type for the variables: string or list.

CoLiS makes an explicit distinction between strings and lists of strings. Since we only have these two kinds of values, we do not use a type system, but can make the distinction on the syntactic level between the categories of string expressions, and the category of list expressions. Consequently, we have two different constructors in the abstract syntax for assignments: one for string values, and one for list values. This separation is made possible by the fact that CoLiS syntax isn't supposed to be written by humans, so that we may simply use different kinds of variables for strings and for lists. The future compiler from shell to CoLiS will reject scripts for which it is not possible to statically infer types of variables and expressions.

Arithmetical expressions, which we could have easily added at this point, are omitted here for the sake of presentation, and since we found that they are very rarely used in our corpus of scripts.

Absence of nested scopes. Note that variables and procedures have to be declared at the beginning of the program, and that the syntax does not provide for nested scopes. This is motivated by the fact that our corpus of scripts only very rarely uses nested shell functions, and that the rare occurrences where they are used in our corpus can easily be rewritten. Hence, we have circumvented the problem of dynamic binding which exists in the shell. The future compiler from shell to CoLiS will reject scripts which make use of dynamic scoping.

Control structures and control flow. Proper handling of exceptions is crucial for shell scripts since in principle any command acting on the file system may fail, and the script should take these possible failures into account and act accordingly. Debian policy even stipulates that fatal errors of commands should usually lead to abortion of the execution of a script, but also allows the maintainer to capture exceptions which he considers as non-fatal. Hence, we have to keep the exception mechanism for the CoLiS language. This decision has an important impact on the semantics of CoLiS, but also shows in the syntax (for instance via the **fatal** term).

The terms **true**, **false**, **return** t and **exit** t correspond to shell built-ins; **fatal** raises a fatal exception which in real shell scripts would be produced by a failing UNIX command. Note that **return** t and **exit** t take a term instead of a natural number. In fact, these commands transform a normal behaviour (of the term t) into an exceptional one for the complete construct. This does only provide for distinction between null or non-null exit codes, which is sufficient for us since we found that the scripts of our corpus very rarely distinguish between different non-null exit codes.

Some shell-specific structures remain. The **shift** command, for instance, removes the first element of the argument list if it exists, and raises an error otherwise.

Values: strings	$\sigma \in \mathit{String}$
Values: lists	$\lambda \in \mathit{StringList} \triangleq \{\sigma^* \mid \sigma \in \mathit{String}\}$
Behaviours: terms	$b \in \{\text{True}, \text{False}, \text{Fatal}, \text{Return True}$ $\text{Return False}, \text{Exit True}, \text{Exit False}\}$
Behaviours: expressions	$\beta \in \{\text{True}, \text{Fatal}, \text{None}\}$
File systems	$\mathcal{FS}$
Environments: strings	$\mathit{SEnv} \triangleq [\mathit{SVar} \rightharpoonup \mathit{String}]$
Environments: lists	$\mathit{LEnv} \triangleq [\mathit{LVar} \rightharpoonup \mathit{StringList}]$
Contexts	$\Gamma \in \mathcal{FS} \times \mathit{String} \times \mathit{StringList} \times \mathit{SEnv} \times \mathit{LEnv}$
Judgments: terms	$t_{/\Gamma} \ \Downarrow \ \sigma \star b_{/\Gamma'}$
Judgments: string fragment	$f_{s/\Gamma} \Downarrow_{sf} \sigma \star \beta_{/\Gamma'}$
Judgements: string expression	$s_{/\Gamma} \Downarrow_{s} \sigma \star \beta_{/\Gamma'}$
Judgements: list fragment	$f_{l/\Gamma} \Downarrow_{lf} \lambda \star \beta_{/\Gamma'}$
Judgements: list expression	$l_{/\Gamma} \Downarrow_{l} \lambda \star \beta_{/\Gamma'}$

Fig. 2. Semantics of CoLiS

Note that the procedure invocation, **call**, does not work on a procedure name with arguments, but on a list whose first element will be considered as the name of the procedure and the remaining part as the arguments. This makes a difference when dealing with empty lists; in that case, the call is a success.

The **pipe** command (the `|` character in shell) takes the standard output of a term and feeds it as input to a second term. The **process** construct corresponds to the invocation of a sub-shell (backquotes, or `$(...)` in shell).

2.3 Semantics

All the elements (that is terms, string fragments and expressions, and list fragments and expressions) of the language are evaluated (see semantic judgements in Figure 2) in a context that contains the file system (left abstract in this work), the standard input, the list of arguments from the command line and the variable environments. They produce a new context, a *behaviour* and a string or a list. In particular, terms produce strings, which is their standard output. For instance, a judgement

$$t_{/\Gamma} \quad \Downarrow \quad \sigma \star b_{/\Gamma'}$$

means that the evaluation of the term t in the context Γ terminates with behaviour b, produces the new context Γ', and the standard output σ.

Note that the file system as well as the built-ins of the shell are left abstract in this work. We focus only on the structure of the language.

$$\frac{}{\mathbf{nil}_{s\,/\varGamma} \Downarrow_{s} \epsilon \star \text{None}_{/\varGamma}} \qquad \frac{f_{s\,/\varGamma} \Downarrow_{sf} \sigma \star \beta_{/\varGamma'} \qquad s_{/\varGamma'} \Downarrow_{s} \sigma' \star \beta'_{/\varGamma''}}{f_s :: s_{/\varGamma} \Downarrow_{s} \sigma \cdot \sigma' \star \beta\beta'_{/\varGamma''}}$$

$$\frac{}{\sigma_{/\varGamma} \Downarrow_{sf} \sigma \star \text{None}_{/\varGamma}} \qquad \frac{}{x_{s\,/\varGamma} \Downarrow_{sf} \varGamma.\texttt{senv}[\texttt{x}_\texttt{s}] \star \text{None}_{/\varGamma}}$$

$$\frac{}{n_{/\varGamma} \Downarrow_{sf} \varGamma.\texttt{args}[\texttt{n}] \star \text{None}_{/\varGamma}} \qquad \frac{t_{/\varGamma} \Downarrow \sigma \star b_{/\varGamma'}}{t_{/\varGamma} \Downarrow_{sf} \sigma \star \overline{b}_{/\varGamma[\texttt{fs}\leftarrow\Gamma'.\texttt{fs};\ \texttt{input}\leftarrow\Gamma'.\texttt{input}]}}$$

Fig. 3. Semantic rules for the evaluation of string expressions and fragments

Behaviours We inherit a quite complex set of possible behaviours of terms from the shell: True, False, Fatal, Return True, Return False, Exit True, Exit False and None. The case of expressions is simpler, their behaviour can only be True for success, Fatal for error, and None for the cases that do not change the behaviour. A term behaviour b can be converted to an expression behaviour $\overline{b}$ as follows:

$$\begin{array}{lll} \overline{b} := & \text{True} & \text{if } b \in \{\text{True}, \text{Return True}, \text{Exit True}\} \\ \mid & \text{Fatal} & \text{otherwise} \end{array}$$

The composition $\beta\beta'$ of two expression behaviours β and β' is defined as :

$$\begin{array}{lll} \beta\beta' := & \beta & \text{if } \beta' = \text{None} \\ \mid & \beta' & \text{otherwise} \end{array}$$

Expressions The semantics of string fragments and expressions are shown in Figure 3. Each expression or fragment is evaluated with respect to a context and produces a value of type string or list, an expression behaviour and a new context. An expression behaviour can be True, Fatal or the absence of behaviour None. Roughly, the behaviour of an expression is the last success or failure of a term observed when evaluating the expression. Expression fragments other than terms do not contribute to the behaviour of a term, this is modeled by giving them the dummy behaviour None.

In the semantics of Figure 3, we write $\varGamma.\texttt{senv}$, $\varGamma.\texttt{lenv}$ and $\varGamma.\texttt{args}$ for the fields of the context $\varGamma$ containing the string environment, the list environments and the argument line respectively.

Figure 4 gives the rules for the evaluation of a "do while" loop, and spells out how the possible behaviours observed when evaluating the condition and the body determine the behaviour of the complete loop.

The pipe construct completely ignores the behaviour of the first term. Finally, the **process** protects part of the context from modifications. Changes to variables and arguments done inside a **process** are not observable. The modifications on the file system and the standard input are kept. Their semantics is given in Figure 5.

$$\frac{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 \in \{\text{Fatal}, \text{Return }\_, \text{Exit }\_\}}{(\textbf{do } t_1 \textbf{ while } t_2)_{/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1}} \text{ \textsc{Transmit-Body}}$$

$$\frac{\begin{array}{c} t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 \in \{\text{True}, \text{False}\} \\ t_{2/\Gamma_1} \Downarrow \sigma_2 \star \text{True}_{/\Gamma_2} \qquad (\textbf{do } t_1 \textbf{ while } t_2)_{/\Gamma_2} \Downarrow \sigma_3 \star b_{3/\Gamma_3} \end{array}}{(\textbf{do } t_1 \textbf{ while } t_2)_{/\Gamma} \Downarrow \sigma_1\sigma_2\sigma_3 \star b_{3/\Gamma_3}} \text{ \textsc{True}}$$

$$\frac{\begin{array}{c} t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 \in \{\text{True}, \text{False}\} \\ t_{2/\Gamma_1} \Downarrow \sigma_2 \star b_{2/\Gamma_2} \qquad b_2 \in \{\text{False}, \text{Fatal}\} \end{array}}{(\textbf{do } t_1 \textbf{ while } t_2)_{/\Gamma} \Downarrow \sigma_1\sigma_2 \star b_{1/\Gamma_2}} \text{ \textsc{False}}$$

$$\frac{\begin{array}{c} t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 \in \{\text{True}, \text{False}\} \\ t_{2/\Gamma_1} \Downarrow \sigma_2 \star b_{2/\Gamma_2} \qquad b_2 \in \{\text{Return }\_, \text{Exit }\_\} \end{array}}{(\textbf{do } t_1 \textbf{ while } t_2)_{/\Gamma} \Downarrow \sigma_1\sigma_2 \star b_{2/\Gamma_2}} \text{ \textsc{Transmit-Cond}}$$

Fig. 4. Semantic rules for the "do while"

$$\frac{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad t_{2/\Gamma_1[\texttt{input}\leftarrow\sigma_1]} \Downarrow \sigma_2 \star b_{2/\Gamma_2}}{\textbf{pipe } t_1 \textbf{ into } t_{2/\Gamma} \Downarrow \sigma_2 \star b_{2/\Gamma_2[\texttt{input}\leftarrow\Gamma_1.\texttt{input}]}} \text{ \textsc{Pipe}}$$

$$\frac{t_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma'}}{\textbf{process } t_{/\Gamma} \Downarrow \sigma \star \bar{b}_{/\Gamma[\texttt{fs}\leftarrow\Gamma'.\texttt{fs},\ \texttt{input}\leftarrow\Gamma'.\texttt{input}]}} \text{ \textsc{Process}}$$

Fig. 5. Semantics of the evaluation for **pipe** and **process**

2.4 Mechanised version

We have formalised the syntax and semantics of CoLiS using the proof environment Why3 [5]. Why3 is an environment dedicated to deductive program verification. It provides both a specification language, and a programming language. The theorems and annotated programs (in fact, everything that needs to be proven) are converted by Why3 into proof obligations and passed to external provers. Its programming language, WhyML, is a language of the ML family containing imperative constructs such as references and exceptions. These elements are well handled in the proof obligations, allowing the user to write programs in a natural way.

The semantics of CoLiS is expressed in the Why3 specification language as a so-called inductive predicate, defined by a set of Horn clauses. The translation is completely straightforward, for instance a fragment of the translation of the semantic rules from Figure 4 to Why3 is shown in Figure 6. Formalising the semantics in Why3 this way has the immediate advantage of syntax and type checks done by the Why3 system, and is of course indispensable for proving the correctness of the interpreter.

```
inductive eval_term term context string behaviour context =

  | EvalT_DoWhile_Transmit_Body : ∀ t₁ Γ σ₁ b₁ Γ₁ t₂.

    eval_term t₁ Γ σ₁ b₁ Γ₁ →
    (match b₁ with BNormal _ → false | _ → true end) →

    eval_term (TDoWhile t₁ t₂) Γ σ₁ b₁ Γ₁

  | EvalT_DoWhile_True : ∀ t₁ Γ σ₁ b₁ Γ₁ t₂ σ₂ Γ₂ σ₃ b₃ Γ₃.

    eval_term t₁ Γ σ₁ (BNormal b₁) Γ₁ →
    eval_term t₂ Γ₁ σ₂ (BNormal True) Γ₂ →
    eval_term (TDoWhile t₁ t₂) Γ₂ σ₃ b₃ Γ₃ →

    eval_term (TDoWhile t₁ t₂) Γ (concat (concat σ₁ σ₂) σ₃) b₃ Γ₃
```

Fig. 6. Why3 version of the semantic rules for the "do while"

3 Interpreter

The interpreter is written in WhyML, the programming language of the Why3 environment, as a set of mutually recursive functions. The functions are written in a standard style combining functional and imperative features. The main interpreter function has the following signature in Why3:

```
let rec interp_term (t: term) (Γ: context) (stdout : ref string)
                                                : (bool, context)
```

There are some fundamental differences between the interpreter on the one hand, and the specification of the semantics on the other hand:

– The function `interp_term` returns normally only in case of normal behaviours.
– The exceptional behaviours Fatal, Return b and Exit b are signaled by raising Why3 exceptions, respectively of the form Fatal(Γ), Return (b, Γ) and Exit (b, Γ) where Γ is the resulting context.
– The standard output is modelled by the mutable variable `stdout` of type string, to which characters are written. This makes the code closer to a standard interpreter which displays results as it produces them.
– The composition of expression behaviours is done by an auxiliary function with an accumulator: instead of yielding the behaviours as a component of a complex result type and then composing them (what corresponds to the semantic rules), we transmit to the recursive call the current behaviour, and let it update it if needed.

To illustrate theses differences we present in Figure 7 an excerpt of the interpreter code for the case of **fatal** command, and the conditional command. Note

```
match t with
| TFatal → raise (EFatal Γ)
| TIf t₁ t₂ t3 →
  let (b₁, Γ₁) =
    try
      interp_term t₁ Γ stdout
    with
      EFatal Γ' → (false, Γ')
    end
  in
  interp_term (if b₁ then t₂ else t3) Γ₁ stdout
...
```

Fig. 7. Code of the interpreter for the **if** construct

that exceptions, other that `EFatal`, potentially raised by the interpretation of t_1 are naturally propagated. This implicit propagation makes the code of the interpreter significantly simpler than the inductive definition of the semantics.

Due to while loops in particular, this interpreter does not necessarily terminate. Yet, we prove that this interpreter is sound and complete with respect to the semantics, as expressed by the two following theorems. We define a notation for executions of the interpreter. For any term t, contexts Γ and Γ', string σ and behaviour b,

$$t_{/\Gamma} \mapsto \sigma \star b_{/\Gamma'}$$

states that when given the term t, the context Γ and a string reference as its input, the interpreter terminates, writing the string σ at the end of the reference. It terminates

- normally when b is True or False, returning the boolean b and the new context Γ';
- with an exception `EFatal`(Γ'), `EReturn`(b', Γ') or `EExit`(b', Γ') when b is Fatal, Return b' or Exit b' respectively.

Theorem 1 (Soundness of the interpreter). *For all t, Γ, σ, b and Γ': if $t_{/\Gamma} \mapsto \sigma \star b_{/\Gamma'}$ then $t_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma'}$*

Theorem 2 (Completeness of the interpreter). *For all t, Γ, σ, b and Γ': if $t_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma'}$ then $t_{/\Gamma} \mapsto \sigma \star b_{/\Gamma'}$*

Due to the mutual recursion in the definition of the abstract syntax, and in the functions of the interpreter, we need of course analogous theorems for string and list fragments and expressions, which are omitted here.

3.1 Proof of soundness

Soundness is expressed in Why3 as a set of post-conditions (see Figure 8) for each function of the interpreter. Why3 handles the recursive functions pretty

```
let rec interp_term (t: term) (Γ: context) (stdout : ref string)
                                                         : (bool, context)
  diverges
  returns { (b, Γ') → ∃ σ. !stdout = concat (old !stdout) σ
                       ∧ eval_term t Γ σ (BNormal b)  Γ' }
  raises  { EFatal Γ' → ∃ σ. !stdout = concat (old !stdout) σ
                       ∧ eval_term t Γ σ BFatal Γ' }
  ...
```

Fig. 8. Contract of the sound interpreter. There are similar post-conditions for other exceptions raised.

well and splits the proof into many simpler sub-goals. However, some of these subgoals still require up to 30 seconds to be proven by the E prover.

One difficulty in the proof comes from the fact that the interpreter uses an additional argument to pass the behaviour of the previous term. This makes the annotations of the functions harder to read and the goals harder to prove, with post-conditions of the form:

```
(eval_sexpr_opt s Γ σ None Γ' ∧ b = previous)
     ∨  eval_sexpr_opt s Γ σ (Some b) Γ
```

for an output (σ, b, Γ') of the expression interpreter.

The choice to have an interpreter with imperative feature (and thus different from the declarative semantics) makes the proof hard. The most disturbing feature for provers is the use of a reference to model the standard output. This causes proof obligations of the form:

```
∃ σ. !stdout = concat (old !stdout) σ ∧ eval_term t Γ σ b Γ'
```

An existential quantification is hard for SMT solvers; it is a challenge for them to find the right instance of the existentially quantified variable that makes the proof work. This is in general a weak point of SMT solvers and requires provers like the E prover which is based on the superposition calculus.

4 Proof of completeness

We show completeness of the interpreter (Theorem 2) by proving two intermediary lemmas. The first lemma states the functionality of our semantic predicates:

Lemma 1 (Functionality of the semantic predicates). *For all t, Γ, Γ_1, Γ_2, σ_1, σ_2, b_1, and b_2: if $t_{/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1}$ and $t_{/\Gamma} \Downarrow \sigma_2 \star b_{2/\Gamma_2}$, then $\sigma_1 = \sigma_2$, $b_1 = b_2$ and $\Gamma_1 = \Gamma_2$.*

This lemma is quite straightforward to prove.

The second lemma states the termination of the interpreter in case one can prove a judgement about the semantics for the same input:

Lemma 2 (Termination of the interpreter). *For all t, Γ, Γ_1, σ_1 and b_1: if $t_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1}$, then the interpreter terminates when given t, Γ.*

It is not obvious how to prove this lemma in the Why3 framework. The difficulty of the proof will be discussed below in Section 4.1, and our solution to the problem is presented in Section 4.2.

Theorem 2, stating the completeness of the interpreter, follows immediately from the above two lemmas, together with Theorem 1 stating the soundness of the interpreter:

Proof. Let t be a term, Γ and Γ_1 contexts, σ_1 a string and b_1 a behaviour. Let us assume that there exists a proof of the judgement $t_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1}$. By Lemma 2 (termination of the interpreter), there exists some results σ_2, b_2 and Γ_2 computed by the interpreter. By Theorem 1 (soundness of the interpreter), we have $t_{/\Gamma} \Downarrow \sigma_2 \star {b_2}_{/\Gamma_2}$. By Lemma 1 (functionality of the semantics), we obtain $\sigma_1 = \sigma_2$, $b_1 = b_2$ and $\Gamma_1 = \Gamma_2$, which allows us to conclude.

4.1 Proving (or not proving) termination with heights and sizes

A first naive idea to prove the two lemmas is to use induction on the structure of the terms. This does, of course, not work since one premise of the rule TRUE for the **do while** construct (see Figure 4) uses the same term as its conclusion.

In fact, what does decrease at every iteration is the proof of the judgement itself. A common way in by-hand proofs to exploit that fact is to use the *size* of the proof (*i.e.* the number of rules involved), or alternatively the *height* of the proof tree.

These numbers could then be passed to the interpreter as a new argument along with a pre-condition specifying that this number corresponds to the size (resp. the height) of the proof. It is then easy to prove that it decreases at each recursive call, and since this value is always positive, we obtain termination of the program. These solutions, however, have drawbacks that make them unsuitable for use in the Why3 environment:

– On the one hand, back-end SMT solvers can reason about arithmetic, but have only incomplete strategies for handling quantifiers; on the other hand superposition provers are good with quantifiers but do not support arithmetic. One could think of replacing an axiomatised arithmetic by a simple successor arithmetic, that is using only *zero* and the *successor* function. This would not solve the problem since when using the size one still needs addition and subtraction, and when using the height one needs the maximum function, and handling of inequalities.
– When we know the size of a proof, we cannot deduce from it the size of the proofs of the premises, which makes the recursive calls complicated.
  A way to solve this problem is to modify the interpreter so that it returns the "unused" size (a technique, sometimes referred to as the *credit* or *fuel*, which can be useful for proving the complexity of a program). This does imply a major modification of the interpreter, though: the exceptions would

```
type skeleton =
  | S0
  | S1 skeleton
  | S2 skeleton skeleton
  | S3 skeleton skeleton skeleton
```

Fig. 9. The data type for skeletons

have to carry that number as well, and the interpreter would have to catch them every time, just to decrement the size and then raise them again.

– We have a similar problem with the height: we cannot deduce from the height of a proof the heights of the premises, but only an upper bound.
  We could solve this problem by using inequalities either in the pre- and post-conditions or in the predicate itself. Nevertheless, it makes the definition of the predicate and the pre- and post-conditions more onerous, and the work of the SMT solvers more complicated.

4.2 Proving termination with ghosts and skeletons

The proof of termination of the interpreter would be easy if we could use an induction on the proof tree of the judgement. The problem is that the proof tree is (implicitly) constructed during the proof, and is not available as a first-class value in the specification. The solution we propose is to modify the predicates specifying the semantics of CoLiS to produce a lightweight representation of the proof tree. This representation, which we call a *skeleton*, contains only the shape of the proof tree. The idea is that a complete proof tree could be abstracted to a skeleton just be ignoring all the contents of the nodes, and just keeping the outline of the tree. This avoids the use of arithmetic, since provers only have to work with a simple algebraic data type.

The definition of the type of skeletons in Why3 is shown in Figure 9. There is one constructor for every number of premises of rules in the definition of the semantics, that is in our case, 0, 1, 2 and 3. We then have alternative definitions of our predicates including their skeleton (see Figure 10).

We can now prove the properties of the semantic predicates by induction on the skeletons. Skeletons make proofs by induction possible when nothing else than the proof is decreasing. In fact, it also has an other interesting advantage: we often need to conduct inductions on our semantic predicates. However, these predicates are mutually recursive and do not work on the same data types, which makes our proofs verbose and annoying. Now, we can run our induction on the skeletons, and that makes the definitions and proofs of the theorems much easier. This is, for instance, the case for the Theorem 1.

It remains the question how to connect the interpreter to the skeletons produced by the predicates. This is where *ghost* arguments come in. In the context of deductive program verification, ghost code [9] is a part of a program that is added solely for the purpose of specification. Ghost code cannot have any impact

```
inductive eval_term term context string behaviour context skeleton =

  | EvalT_DoWhile_True : ∀ t1 Γ σ1 b1 Γ1 t2 σ2 Γ2 σ3 b3 Γ3 s1 s2 s3.

    eval_term t1 Γ σ1 (BNormal b1) Γ1 s1 →
    eval_term t2 Γ1 σ2 (BNormal True) Γ2 s2 →
    eval_term (TDoWhile t1 t2) Γ2 σ3 b3 Γ3 s3 →

    eval_term (TDoWhile t1 t2) Γ
              (concat (concat σ1 σ2) σ3) b3 Γ3 (S3 s1 s2 s3)
```

Fig. 10. (Part of the) predicates with skeletons

```
let rec interp_term (t: term) (g: context) (stdout : ref string)
                                    (ghost sk: skeleton) : (bool, context)

  requires { ∃ s b g'. eval_term t g s b g' sk }
  variant { sk }
  returns { (b, g') → ∃ s. !stdout = concat (old !stdout) s
                              ∧ eval_term t g s (BNormal b)  g' sk }
```

Fig. 11. Contract for the terminating interpreter. There are similar post-conditions for exceptions raised.

on the execution of the code: it must be removable without any observable difference on the program. In this spirit, we extend the functions of the interpreter with a ghost parameter which holds the skeleton (see Figure 11).

We also add ghost code in the body of the function (see Figure 12) in order to give indications to the provers, using some auxiliary destructor functions for skeletons. The function `skeleton23`, for instance, takes a skeleton that is required to have a head of arity 2 or 3, and returns its direct subtrees. The fact that it *requires* the skeleton to have a head of arity 2 or 3 adds the right axioms and goals to the proof context, thus helping the provers.

This works well because we wrote the semantics in a specific way: the order of the premises always corresponds to the order in which the computation must happen. This means that we can take the skeleton of the first premise and give it to the first recursive call. After that call, either an exception is raised which interrupts the control flow, or we take the skeleton of the second premise and give it to the second recursive call.

It would have been tempting to match on the term and the skeleton at the same time (to have something like `|TDoWhile t1 t2, S1 sk1 → .`). This, however, does not work, since it would make the execution of the code dependent on a ghost parameter, which is rejected by the type checker of Why3 as a forbidden effect of ghost code on non-ghost code [9].

```
| TDoWhile t₁ t₂ →
  let ghost sk1 = skeleton123 sk in
  (* At this point, we know that the rules
     that might apply can have 1, 2 or 3 premises. *)
  let (b₁, g₁) = interp_term t₁ g stdout sk1 in
  let (b₂, g₂) =
    try
      let ghost (_, sk2) = skeleton23 sk in
      (* At this point, we know that the rule
         with 1 premise cannot be applied anymore. *)
      interp_term t₂ g₁ stdout sk2
    with
      EFatal g₂ → (false, g₂)
    end
  in
  if b₂ then
    let ghost (_, _, sk3) = skeleton3 sk in
    (* And finally, only the rule with 3 premises can be applied. *)
    interp_term (TDoWhile t₁ t₂) g₂ stdout sk3
  else
    (b₁, g₂)
```

Fig. 12. Body of the terminating interpreter

4.3 Reproducibility

Using the technique of skeletons, all the proof obligations are proven by automated provers. The proof takes some time because there are a many cases (we obtain 207 subgoals), but none of those takes more than 4 seconds to our provers.

The Why3 code for the syntax, semantics, the interpreter and all the proofs is available online [13]. The proofs need of course Why3 [5], and at least the provers Alt-Ergo [4] (1.30), Z3 [16] (4.5.0) and E [17] (1.9.1). One may in addition use CVC3 [3], CVC4 [2] and SPASS [19] in order to gain additional confirmation.

5 Related work

Formalising the semantics of programming language is most of the time done using interactive proof assistants like Coq or Isabelle. Yet, formalising semantics and proving complex properties, with automatic provers only, was already shown possible by Clochard et. al [8], who also use the Why3 environment. The difficulty of proving completeness was not addressed in that work, though. Interestingly, the issues we faced regarding completeness and inductive predicates was present in other work conducted within the CoLiS project by Chen et al. [7], for proving a shell path resolution algorithm. They solve completeness by indexing their inductive predicates with heights. We would like to investigate whether an approach with skeletons instead of heights would make the proofs easier. To our knowledge, the idea of using proof skeletons is new, even though the idea seems quite close to the concept of step-indexing for reasoning on operational semantic rules [1].

Several tools can spot certain kinds of errors in shell scripts. The tool `checkbashisms` [6], for instance, detects usage of bash-specific constructs in shell scripts. It is based on regular expressions. The `ShellCheck` [11] tool detects error-prone usages of the shell language. This tool is written in Haskell and analyses the scripts on-the-fly while parsing.

There have been few attempts to formalize the shell. Recently, Greenberg [10] has presented elements of formal semantics of POSIX shell. The work behind Abash [15] contains a formalization of the part of the semantics concerned with variable expansion and word splitting. The Abash tool itself performs abstract interpretation to analyze possible arguments passed by Bash scripts to UNIX commands, and thus to identify security vulnerabilities in Bash scripts.

6 Conclusion and future work

We presented a Why3 implementation of the semantics of an imperative programming language. This formalisation is faithful to the semantic rules written by hand. Our main contribution is an interpreter for this language proven both sound and complete. The proof of completeness uses an original technique involving what we call skeletons: an abstraction of the proof tree for an inductive predicate, that decreases on recursive call, allowing us to use induction on the proof itself.

Future work In the near future, we would like to try a more direct proof of completeness (*i.e.* without separating it into the soundness, the functionality of the semantic predicates and the termination of the algorithm). Such a proof would be interesting in cases where the functionality can not be proven (when we can derive the same judgement in different manners, for instance).

To fulfil our mid-term goal to verify shell scripts in Debian packages, we will need to formalise the file system as well as its built-ins. We will also have to write the automated translation from shell to CoLiS. This translation will have to analyse the scripts statically to determine, among other things, the type of the variables. The first step of this compiler, the parser of POSIX shell scripts, is described in [14].

References

1. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. Program. Lang. Syst. 23(5), 657–683 (Sep 2001), `http://doi.acm.org/10.1145/504709.504712`
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proceedings of the 23rd international conference on Computer aided verification. pp. 171–177. CAV'11, Springer-Verlag, Berlin, Heidelberg (2011), `http://cvc4.cs.stanford.edu/web/`

3. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) 19th International Conference on Computer Aided Verification. Lecture Notes in Computer Science, vol. 4590, pp. 298–302. Springer, Berlin, Germany (Jul 2007)
4. Bobot, F., Conchon, S., Contejean, E., Iguernelala, M., Lescuyer, S., Mebsout, A.: The Alt-Ergo automated theorem prover (2008), `https://alt-ergo.ocamlpro.com/`
5. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64. Wrocław, Poland (August 2011), `http://proval.lri.fr/publications/boogie11final.pdf`
6. Braakman, R., Rodin, J., Gilbey, J., Hobley, M.: checkbashisms, `https://sourceforge.net/projects/checkbaskisms/`
7. Chen, R., Clochard, M., Marché, C.: A formal proof of a unix path resolution algorithm. Research Report RR-8987, Inria Saclay Ile-de-France (Dec 2016)
8. Clochard, M., Filliâtre, J.C., Marché, C., Paskevich, A.: Formalizing semantics with an automatic program verifier. In: Giannakopoulou, D., Kroening, D. (eds.) 6th Working Conference on Verified Software: Theories, Tools and Experiments (VSTTE). Lecture Notes in Computer Science, vol. 8471, pp. 37–51. Springer, Vienna, Austria (2014)
9. Filliâtre, J.C., Gondelman, L., Paskevich, A.: The Spirit of Ghost Code. In: CAV 2014, Computer Aided Verification - 26th International Conference. Vienna Summer Logic 2014, Austria (Jul 2014), `https://hal.inria.fr/hal-00873187`
10. Greenberg, M.: Understanding the POSIX shell as a programming language. In: Off the Beaten Track 2017. Paris, France (Jan 2017)
11. Holen, V.: Shellcheck, `https://github.com/koalaman/shellcheck`
12. IEEE and The Open Group: POSIX.1-2008/Cor 1-2013, `http://pubs.opengroup.org/onlinepubs/9699919799/`
13. Jeannerod, N.: Full Why3 code for the CoLiS language and its proofs, `http://toccata.lri.fr/gallery/colis_interpreter.en.html`
14. Jeannerod, N., Régis-Gianas, Y., Treinen, R.: Having Fun With 31.521 Shell Scripts (Apr 2017), `https://hal.archives-ouvertes.fr/hal-01513750`, working paper
15. Mazurak, K., Zdancewic, S.: ABASH: finding bugs in bash scripts. In: PLAS07: Proceedings of the 2007 workshop on Programming languages and analysis for security. pp. 105–114. San Diego, CA, USA (Jun 2007)
16. de Moura, L., Bjørner, N.: Z3, an efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008), `https://github.com/Z3Prover/z3`
17. Schulz, S.: System description: E 0.81. In: Basin, D.A., Rusinowitch, M. (eds.) Second International Joint Conference on Automated Reasoning. Lecture Notes in Computer Science, vol. 3097, pp. 223–228. Springer (2004), `http://wwwlehre.dhbw-stuttgart.de/~sschulz/E/E.html`
18. The Debian Policy Mailing List: Debian Policy Manual, `https://www.debian.org/doc/debian-policy/`
19. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) 22nd International Conference on Automated Deduction. Lecture Notes in Computer Science, vol. 5663, pp. 140–145. Springer (2009), `http://www.mpi-inf.mpg.de/departments/automation-of-logic/software/spass-workbench/classic-spass-theorem-prover/`