Le coquillage dans le CoLiS*-mateur
Formalisation d'un langage de programmation de type shell

Nicolas Jeannerod

Résumé

Le langage shell est largement utilisé pour l'installation des paquets logiciel dans les distributions Unix. Notre objectif à moyen terme est d'analyser la robustesse et la correction de tels scripts. Toutefois, la syntaxe et la sémantique du shell sont particulièrement piégeuses. Une description formelle du langage utilisé est donc un préalable à l'analyse de ces scripts.

Nous avons identifié un sous-langage du shell utilisé dans le corpus des scripts d'installation de Debian. Nous proposons un nouveau langage – nommé CoLiS – conçu pour permettre dans le futur une traduction automatique des scripts de notre corpus. Dans cet article, nous présentons une formalisation mécanisée de la syntaxe et de la sémantique de CoLiS ainsi qu'un interpréteur de référence dont la correction est prouvée avec l'environnement de preuve de programmes Why3.

1. Introduction

Le shell – ou plus exactement le shell *Unix* – est un interpréteur de commandes apparu en 1971 [1] pour la première version d'Unix. De nombreuses versions ont été développées, et la version sans doute majoritairement utilisée aujourd'hui – notamment dans Debian – est le *Bourne-Again shell* [2] (bash). La version du shell que nous considérerons dans ce papier, et que nous nous permettrons d'appeler simplement "shell" par la suite, est celle qui est décrite dans la *Debian Policy [1], section 10.4, Scripts*. Il s'agit pour l'essentiel du shell décrit par le standard POSIX [2].

Dans cet article, nous nous intéressons particulièrement aux scripts shell dans le cadre de l'installation et la désinstallation des paquets dans la distribution Debian. Un paquet y est composé de plusieurs éléments :

- Une archive contenant des fichiers à répartir dans le système de fichiers. Ce peuvent être des fichiers de configuration par défaut, les binaires d'une application, etc.
- Un certain nombre de fichiers donnant des méta-informations sur le paquet comme ses dépendances, sa version, son auteur, sa description, etc.
- Un certain nombre de scripts chargés de préparer le système avant une installation ou une mise à jour ou de nettoyer le système après une désinstallation. Dans la majorité [3] des cas, ces scripts sont des scripts shell. Ils sont exécutés avec les privilèges maximaux – c'est-à-dire en tant que l'utilisateur *root* – ce qui rend la moindre erreur potentiellement désastreuse.

∗. Ce travail a partiellement été supporté par le projet ANR CoLiS (contrat ANR-15-CE25-0001).

1. Il s'agissait alors du *Thompson shell*, du nom de Ken Thompson, son auteur et l'un des créateurs d'Unix.
2. Écrit en 1988 par Brian Fox dans le cadre du projet GNU.
3. En mai 2016, 98.9% des scripts d'installation de Debian stable étaient des scripts shell.

```
f () { g "$@"; }
g () { echo "$1 $a $b"; }
a=foo
f $a
```

(a) Tout est global et dynamique

```
x=
! true
$x
```

(b) L'appel de fonction vide

```
x="echo foo"
$x && "$x"
```

(c) Des appels de fonction vicieux

```
x="foo $(false) bar" || echo $x
```

(d) L'expansion

```
! true
false && true
echo foo
false
echo bar
```

(e) Le faux et le fatal

```
echo "$(return 0)foo"
exit 2 | echo bar
( exit 3 ) || echo baz
```

(f) La dissimulation des erreurs

FIGURE 1 – Des scripts shell

L'installation d'un paquet est gérée par dpkg, un logiciel à la base du système de paquets de Debian. Une installation consiste à lancer un script de pré-installation dans un premier temps, extraire l'archive dans le système de fichiers dans un deuxième temps et exécuter un script de post-installation dans un dernier temps. La moindre erreur d'exécution interrompt le procédé. On déclare dans ce cas que l'installation a été un échec. La désinstallation et la mise à jour sont similaires.

Ce procédé d'installation est très sensible à la moindre erreur se trouvant dans un script shell. C'est à cause de cet enjeu qu'il est intéressant de produire un outil automatique de vérification des scripts d'installations de Debian. Cet outil devrait être capable de détecter les scripts dangereux, et d'indiquer la cause de cette dangerosité. Dans le cadre général, la vérification de scripts shell est inenvisageable. Cependant, la restriction aux paquets de Debian donne bon espoir. En effet, les scripts d'installation sont des scripts souvent simples et répétitifs. Ils sont écrits par les administrateurs de paquets, qui ont – en général – une certaine connaissance du shell, évitent les mauvaises pratiques, et sont sensibles aux arguments de la vérification. Le cadre de l'exécution de ces scripts restreint également les propriétés à vérifier ; il n'y a par exemple pas de problème de concurrence. De plus, la Debian Policy a des exigences qui guident le choix des propriétés que l'on veut imposer aux scripts, évitant aux membres du projet CoLiS de décider pour d'autres ce qui est acceptable et ce qui ne l'est pas.

Même dans ce cadre, la syntaxe et la sémantique du shell posent problème, car elles peuvent être extrêmement piégeuses à la fois pour le développeur et pour les outils d'analyse. En témoignent les scripts d'exemple donnés en figure 1 :

- La figure 1a met en évidence que tout est dynamique : le nom d'une fonction ou variable n'est résolu qu'au moment de son utilisation. Dans cet exemple, la sortie sera "foo␣foo␣".
- Les figures 1b et 1c présentent des subtilités de l'exécution des commandes simples [4]. Dans la figure 1b, le résultat sera un code de retour nul, et ce même si la commande

4. Voir *POSIX, Shell and Utilities, 2.9.1 Simple Commands.*

précédente a un code de retour non nul. Dans la figure 1c, la sortie sera “foo” avec un code retour de 127 indiquant que la commande `echo foo` n’a pas été trouvée.

- La figure 1d donne deux exemples d’expansion, l’un dans l’affectation, et l’autre dans l’appel de `echo`. Ici, la sortie sera “foo␣␣bar”.
- La figure 1e, quant à elle, met en évidence la différence entre le `! true`, le `false && true` et le `false` due à l’utilisation du mode strict [5]. La sortie standard sera “foo” avec un code de retour de 1.
- Enfin, la figure 1f montre diverses façons de dissimuler des comportements exceptionnels, comme produits par `return` ou `exit`. La sortie standard sera “foo␣bar␣baz”.

Dans cet article, nous proposons un langage intermédiaire – nommé CoLiS – dont la conception est guidée par les objectifs suivants :

- Il doit être « plus propre » que le shell : on en enlève des structures dangereuses – comme le `eval` qui permet d’exécuter n’importe quel code donné sous forme de *string* – et on rend plus explicite les dangers qu’on ne peut pas éliminer.
- Il doit avoir une syntaxe et une sémantique claires. L’objectif étant à la fois d’aider les outils d’analyse dans leur travail, et de permettre à un lecteur de pouvoir se convaincre de la correction de ces outils sans devoir se préoccuper des pièges de la syntaxe ou de la sémantique du langage sous-jacent.
- Sa sémantique se doit d’être moins dynamique que celle du shell. Cela peut être atteint à l’aide d’une certaine discipline de typage avec, par exemple, l’obligation de déclarer ses variables et fonctions dans une en-tête.
- Il doit être prévu pour être la cible d’une traduction automatique depuis shell. La traduction pouvant difficilement être prouvée [6], il faudra pouvoir se convaincre de sa correction par sa lecture ou des tests. Pour cette raison, le langage CoLiS ne pourra pas être trop éloigné du shell.

Ce langage n’est en aucun cas conçu comme un remplacement du shell dans les paquets logiciels. Il conserve de nombreux traits – et défauts – du shell et n’apporterait par conséquent pas tant de nouveauté.

Notre objectif à moyen terme sera d’analyser des scripts shell traduits dans notre langage CoLiS et de vérifier des propriétés telles que :

- L’exécution se déroule sans erreur ;
- L’exécution conserve un invariant, comme une structure particulière du système de fichiers ;
- Deux scripts commutent ;
- Un script est idempotent, ce qui est requis par la Debian Policy [7] ;
- Un script est l’inverse d’un autre, par exemple pour les scripts d’installation et de désinstallation – en fait, de purge – : « removal-scripts $\circ$ install-scripts = identity ».

5. C’est le mode activé par la commande `set -e` ou par l’appel de l’interpréteur avec l’option `-e`. Ce mode est très fortement conseillé par la Debian Policy. Voir *Debian Policy [1], section 6.1, Introduction to package maintainer scripts* pour cette requête, et *POSIX, Shell and Utilities, 2.14 Special Built-In Utilities* pour l’explication de son effet.

6. Si l’on pouvait avoir une syntaxe et une sémantique propres pour le shell, nous n’aurions pas besoin de ce langage CoLiS, et *a fortiori* de cette traduction.

7. Il ne s’agit pas ici de l’idempotence au sens mathématique, mais plutôt de la capacité à récupérer sur erreur. Voir *Debian Policy [1], section 6.2, Maintainer scripts idempotency.*

Dans ce papier, nous présentons en section 2 le langage CoLiS, sa syntaxe – abstraite – et une partie de sa sémantique. La section 3 est consacrée à une formalisation dans l'environnement de vérification Why3 de cette syntaxe et de cette sémantique accompagnées d'un interpréteur – prouvé correct – pour le langage. Nous comparons ensuite avec d'autres travaux en section 4 et concluons en section 5.

2. Le langage CoLiS

La syntaxe et la sémantique du langage CoLiS sont présentées en figures 3 et 4 respectivement. Il s'agit d'une syntaxe abstraite, car le langage n'est pas destiné à être utilisé – du moins tel quel – par des humains. À titre d'exemple, on trouvera des traductions des scripts shell de la figure 1 en figure 2. Nous allons maintenant parcourir les éléments du langage, les expliquer et présenter la sémantique de certains d'entre eux.

CoLiS est un langage impératif avec quelques structures complexes comme les *string* et *list expressions*. Sa complexité vient – en grande partie – de sa proximité à shell. En l'occurrence, ses expressions visent à représenter le mécanisme très complexe de l'expansion du shell.

Tous les éléments du langage s'évaluent dans un contexte, lequel comprend le système de fichiers – laissé abstrait dans tout ce travail –, l'entrée standard, la ligne d'arguments et les environnements de fonctions et de variables de types *string* et liste [8]. Ils produisent un nouveau contexte et une *string* ou une liste selon les cas : les expressions de type *string* s'évaluent en des *strings* ; celles de type liste en des listes ; et les termes produisent des *strings* – leurs sorties standards.

2.1. Variables et expressions

On trouve deux types possibles pour les variables : *string* ou liste. Ces deux types correspondent à deux usages des variables du shell. Le premier usage consiste à employer les variables pour stocker des *strings* – un nom de fichier par exemple. Le second consiste à utiliser le mécanisme du shell qui découpe les variables au niveau des espaces pour stocker des listes et non plus des *strings*. Ce dernier usage est illustré en figure 5 où l'on construit la liste des arguments d'une commande `ls` dynamiquement.

```
args='-l -a'
if $HUMAN_READABLE
then
  args="$args -h"
fi
ls $args /
```

FIGURE 5 – Variable shell utilisée comme une liste

La distinction entre ces types apparaît explicitement dans la syntaxe du langage CoLiS. On a par exemple deux affectations différentes :

$$x_s := s \qquad\qquad x_l := l$$

Cette distinction est rendue possible d'une part par le fait que CoLiS ne vise pas à être utilisé par des humains, et d'autre part parce que le traducteur automatique des scripts shell vers les scripts CoLiS que nous prévoyons d'écrire pourra déterminer ces types à l'aide d'une analyse statique.

8. Dans tout cet article, les listes seront des listes de *strings*.

```
f () {
  echo "$1 $a"
}

a=foo
f $a
```

```
varstring a
proc f is (
  call ( split "echo" , [ arg 1 . "␣" . var a ] )
)
program (
  a :=s "foo" ;
  call ( split "f" , split ( var a ) )
)
```

```
x="foo $(false) bar" \
  || echo $x
```

```
varstring x
program (
  if ( x :=s "foo␣" . term false . "␣bar" ) then
    true
  else
    call ( split "echo" , split (var x) )
)
```

```
x="echo foo"
$x && "$x"
```

```
varstring x
program (
  x :=s "echo␣foo"
  if ( call ( split ( var x ) ) ) then
    call ( var x )
  else
    false
)
```

```
x=
false && true
$x
```

```
varstring x
program (
  x :=s ;
  if fatal then true else false ;
  call ( split (var x) )
)
```

La syntaxe concrète utilisée ici pour CoLiS correspond approximativement à la syntaxe abstraite présentée en figure 3. Elle utilise quelques mots clefs supplémentaires ainsi que les parenthèses afin d'éviter les ambiguïtés.

FIGURE 2 – Des scripts shell et leurs traductions en CoLiS

Variables : *strings*	x_s	$\in$	$SVar$
Variables : listes	x_l	$\in$	$LVar$
Noms de procédures	c	$\in$	$\mathcal{F}$
Entier naturel	n	$\in$	$\mathbb{N}$
Strings	σ	$\in$	$String$
Programmes	p	$::=$	$vdecl^*\ pdecl^*$ **program** t
Déclarations : variables	$vdecl$	$::=$	**varstring** x_s | **varlist** x_l
Déclarations : procédures	$pdecl$	$::=$	**proc** c **is** t
Expressions : *strings*	s	$::=$	f_s^*
Fragments : *strings*	f_s	$::=$	σ | x_s | n | t
Expressions : listes	l	$::=$	f_l^*
Fragments : listes	f_l	$::=$	$[s]$ | **split** s | x_l
Termes	t	$::=$	**true** | **false** | **fatal**
		|	**return** t | **exit** t
		|	$x_s := s$ | $x_l := l$
		|	t ; t | **if** t **then** t **else** t
		|	**for** x_s **in** l **do** t | **do** t **while** t
		|	**process** t | **pipe** t **into** t
		|	**call** l | **shift**

FIGURE 3 – Syntaxe de CoLiS

Valeurs : *strings*	σ	$\in$	$String$
Valeurs : listes	λ	$\in$	$StringList \triangleq \{\sigma^* \mid \sigma \in String\}$
Comportements : termes	b	$\in$	$\{\text{True}, \text{False}, \text{Fatal}, \text{Return True}$
			$\text{Return False}, \text{Exit True}, \text{Exit False}\}$
Comportements : expressions	β	$\in$	$\{\text{True}, \text{Fatal}, \text{None}\}$
Systèmes de fichiers			$\mathcal{FS}$
Environnements : *strings*			$SEnv \triangleq [SVar \to String]$
Environnements : listes			$LEnv \triangleq [LVar \to StringList]$
Contextes	Γ	$\in$	$\mathcal{FS} \times String \times StringList \times SEnv \times LEnv$
Jugements : termes			$t_{/\Gamma} \quad \Rightarrow \quad \sigma \star b_{/\Gamma'}$
Jugements : *string fragment*			$f_{s/\Gamma} \quad \rightsquigarrow_{sf} \quad \sigma \star \beta_{/\Gamma'}$
Jugements : *string expression*			$s_{/\Gamma} \quad \rightsquigarrow_{s} \quad \sigma \star \beta_{/\Gamma'}$
Jugements : *list fragment*			$f_{l/\Gamma} \quad \rightsquigarrow_{lf} \quad \lambda \star \beta_{/\Gamma'}$
Jugements : *list expression*			$l_{/\Gamma} \quad \rightsquigarrow_{l} \quad \lambda \star \beta_{/\Gamma'}$

FIGURE 4 – Sémantique de CoLiS

$$\frac{}{\sigma_{/\Gamma} \rightsquigarrow_{sf} \sigma \star \mathrm{None}_{/\Gamma}}$$

$$\frac{}{x_{s/\Gamma} \rightsquigarrow_{sf} \Gamma.senv[x_s] \star \mathrm{None}_{/\Gamma}}$$

$$\frac{}{n_{/\Gamma} \rightsquigarrow_{sf} \Gamma.args[n] \star \mathrm{None}_{/\Gamma}}$$

$$\frac{t_{/\Gamma} \Rightarrow \sigma \star b_{/\Gamma'}}{t_{/\Gamma} \rightsquigarrow_{sf} \sigma \star \overline{b}_{/\Gamma'}}$$

(a) *String fragments*

$$\frac{}{\varepsilon_{s/\Gamma} \rightsquigarrow_{s} \text{“”} \star \mathrm{None}_{/\Gamma}}$$

$$\frac{f_{s/\Gamma} \rightsquigarrow_{sf} \sigma \star \beta_{/\Gamma'} \qquad s_{/\Gamma'} \rightsquigarrow_{s} \sigma' \star \beta'_{/\Gamma''}}{f_s s_{/\Gamma} \rightsquigarrow_{s} \sigma \cdot \sigma' \star \beta\beta'_{/\Gamma''}}$$

(b) *String expressions*

$$\frac{s_{/\Gamma} \rightsquigarrow_{s} \sigma \star \beta_{/\Gamma'}}{[s]_{/\Gamma} \rightsquigarrow_{lf} [\sigma] \star \beta_{/\Gamma'}}$$

$$\frac{s_{/\Gamma} \Rightarrow \sigma \star \beta_{/\Gamma'}}{\textbf{split}\ s_{/\Gamma} \rightsquigarrow_{lf} \textbf{split}\ \sigma \star \beta_{/\Gamma'}}$$

$$\frac{}{x_{l/\Gamma} \rightsquigarrow_{lf} \Gamma.lenv[x_l] \star \mathrm{None}_{/\Gamma}}$$

(c) *List fragments*

$$\frac{}{\varepsilon_{l/\Gamma} \rightsquigarrow_{l} [] \star \mathrm{None}_{/\Gamma}}$$

$$\frac{f_{l/\Gamma} \rightsquigarrow_{lf} \lambda \star \beta_{/\Gamma'} \qquad l_{/\Gamma'} \rightsquigarrow_{l} \lambda' \star \beta'_{/\Gamma''}}{f_l l_{/\Gamma} \rightsquigarrow_{l} \lambda{+\!\!+}\lambda' \star \beta\beta'_{/\Gamma''}}$$

(d) Listes

FIGURE 6 – Sémantique de l'évaluation des *strings* et listes

De plus, les expressions sont séparées syntaxiquement entre *strings* et listes. Leur syntaxe et leur sémantique restent cependant très similaires : les expressions sont des listes de fragments, et l'évaluation des expressions est la concaténation – de *strings* ou de listes respectivement – des évaluations des fragments. Seules les définitions syntaxiques et sémantiques de ces fragments diffèrent : pour les *string fragments*, ils comprennent les constantes et les variables de type *string*, les éléments de la ligne d'arguments et les termes ; pour les listes, ils comprennent les variables de listes et les *string expressions* vues comme des singletons – ce que permettent les guillemets doubles du shell – ou comme des *strings* à découper – ce que permet l'absence de guillemets doubles en shell.

La sémantique de ces expressions est présentée en figure 6. Chaque expression ou fragment s'évalue dans un contexte et produit une valeur *string* ou une liste, un comportement d'expression et un nouveau contexte. Un comportement d'expression peut être True, Fatal ou l'absence de comportement None. Le comportement d'expression se charge de retenir le dernier succès ou la dernière erreur d'exécution d'un terme au milieu d'une expression. Tout les fragments ne changent donc pas le comportement de l'expression ; d'où l'existence de cette absence de comportement None.

Dans la sémantique de la figure 6, on note ε_s et ε_l les *string* et *list expressions* vides respectivement, “” et [] la *string* et la liste vides et $\cdot$ et $+\!\!+$ les concaténations de *strings* et listes. Pour un contexte Γ, on note $\Gamma.senv$, $\Gamma.lenv$ et $\Gamma.args$ ses composantes contenant l'environnement des *strings*, l'environnement des listes et la ligne d'arguments respectivement.

Pour deux comportements d'expression β et β', on note $\beta\beta'$ leur composition :

$$\begin{array}{rcl} \beta\beta' & := & \beta \text{ si } \beta' = \text{None} \\ & | & \beta' \text{ sinon} \end{array}$$

On note **split** la fonction qui découpe une *string* aux espaces et renvoie ses parties dans une liste. Enfin, pour un comportement de terme b, on note $\overline{b}$ le comportement d'expression associé :

$$\begin{array}{rcl} \overline{b} & := & \text{True si } b \in \{\text{True}, \text{Return True}, \text{Exit True}\} \\ & | & \text{Fatal sinon} \end{array}$$

2.2. Comportements de termes

Les cinq premiers termes présentés, **true**, **false**, **fatal**, **return** t et **exit** t, correspondent aux *built-ins* du shell `true`, `false`, `return` et `exit`. Trois choses sont cependant à noter :

- Les structures **return** t et **exit** t prennent un terme et non pas un entier. En fait, ces commandes se chargent de transformer un comportement normal en comportement exceptionnel. Cela permet d'encoder la plupart des usages des structures du shell `return` et `exit`. En effet, il est très rare [9] de voir l'utilisation de l'arithmétique dans un `return` ou un `exit` ; les usages courants sont plutôt de renvoyer une constante ou de transformer le code de retour de la commande précédente en un comportement exceptionnel.
- On n'a plus seulement `false` mais **false** et **fatal**. Cela vient de l'obligation par la Debian Policy d'utiliser le « mode strict » dans ses scripts [10]. On obtient alors deux comportements « faux » – c'est-à-dire avec un code de retour non nul – selon l'endroit où l'on se trouve. C'est pour mettre en évidence la différence entre ces deux comportements qu'ont été introduites ces deux structures distinctes.
- Tous les codes de retour non nuls du shell sont confondus [11].

Dans le shell, les comportements possibles sont le code de retour nul indiquant le succès, les codes de retour non nuls indiquant l'échec – et qui, selon l'endroit où ils se trouvent se comportent différemment –, le `return` des fonctions avec un code de retour nul ou non et le `exit` avec un code de retour nul ou non. Cela nous donne donc sept comportements : True, False, Fatal, Return True, Return False, Exit True et Exit False. On souhaiterait pouvoir ranger ces comportements dans des catégories plus grandes, comme les comportements normaux et les comportements exceptionnels. Ce n'est malheureusement pas possible, car chaque structure du shell a sa façon à elle de traiter les comportements [12], comme on peut le voir en figure 7.

9. L'arithmétique est utilisée dans moins de 0.2% des scripts.
10. C'est le mode activé par la commande `set -e` ou par l'appel de l'interpréteur avec l'option `-e`. Ce mode est très fortement conseillé par la Debian Policy. Voir *Debian Policy [1], section 6.1, Introduction to package maintainer scripts* pour cette requête, et *POSIX, Shell and Utilities, 2.14 Special Built-In Utilities* pour l'explication de son effet.
11. Cela est justifié par les statistiques : moins de 0.3% des scripts distinguent les codes de retour non nuls.
12. On pourrait imaginer des structures plus bas niveau permettant de factoriser ces comportements. Cependant, cela éloignerait CoLiS du shell.

	True	False	Fatal	Return True	Return False	Exit True	Exit False
Pipe	Normal						
Séquence $(t; t)$	Normal		Exception				
Test (**if**, **do while**)	Vrai	Faux	Exception				
Appel de fonction (**call**)	Succès	Échec	Succès	Échec	Exception		
Sous-processus (**process**)	Succès	Échec	Succès	Échec	Succès	Échec	

FIGURE 7 – Groupement des comportements par les structures de contrôle

2.3. Structures de contrôle classiques

On retrouve dans CoLiS des structures de contrôle classiques : la composition séquentielle t ; t, le **if** t **then** t **else** t permettant de faire un branchement en fonction de l'évaluation d'un terme, le **for** x_s **in** l **do** t permettant d'itérer sur une liste et le **do** t **while** t permettant de boucler jusqu'à ce que l'évaluation d'un terme décide de l'arrêt.

Toute la différence entre le mode « normal » et le mode strict se trouve dans la sémantique de la séquence, car c'est la seule structure qui distingue False et Fatal. Dans le mode strict, la séquence est interrompue par les comportements Return b, Exit b mais également Fatal, alors qu'elle considère comme normaux True et False. Dans le mode « normal », la séquence n'est interrompue que par les Return b et Exit b alors qu'elle considère comme normaux True, False et Fatal.

On notera la présence du **do while** et non pas du **while**. Cela vient du fait que la sémantique du premier est plus simple à décrire. En effet, le **while** a la particularité que, lorsqu'on n'entre jamais dans la boucle – quand le test est d'emblée faux –, il doit renvoyer un comportement True [13]. Cela obligerait donc à traiter spécifiquement le premier test, ajoutant par là trois règles à la sémantique. Étant donné que le **while** peut être obtenu en combinant le **if** avec le **do while**, il a été éliminé en faveur de ce dernier.

On notera également l'absence de certaines structures familières aux utilisateurs du shell, comme le « non » `!`, le « et » `&&` et le « ou » `||`. Toutes ces structures peuvent être obtenues par des combinaisons simples de **if**, **true** et **false** [14] :

$$\begin{aligned} t_1 \texttt{ \&\& } t_2 &\quad - \quad \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else false} \\ t_1 \texttt{ || } t_2 &\quad - \quad \textbf{if } t_1 \textbf{ then true else } t_2 \\ \texttt{! } t &\quad - \quad \textbf{if } t \textbf{ then false else true} \end{aligned}$$

2.4. Processus et fonctions

Restent quelques structures plus spécifiques au shell. Le **shift**, par exemple, permet – en CoLiS comme en shell – de décaler la liste des arguments en supprimant le premier, s'il existe. S'il n'existe pas, on a affaire à un comportement indéfini [15].

13. Voir *POSIX, Shell and Utilities, 2.9.4 Compound Commands* à propos du **while** : "The exit status of the while loop shall be the exit status of the last [body] executed, or zero if none was executed."

14. Ce n'est pas aussi simple en shell, à cause des codes de retour et du mode strict.

15. Voir *POSIX, Shell and Utilities, 2.14 Special Built-In Utilities* : "If the n operand is invalid or is greater than "$#", this may be considered a syntax error and a non-interactive shell may exit ; if the shell does not

$$\frac{t_{1/\Gamma} \Rightarrow \sigma_1 \star b_{1/\Gamma_1} \quad t_{2/\Gamma_1[input \leftarrow \sigma_1]} \Rightarrow \sigma_2 \star b_{2/\Gamma_2}}{\textbf{pipe}\ t_1\ \textbf{into}\ t_{2/\Gamma} \Rightarrow \sigma_2 \star b_{2/\Gamma_2[input \leftarrow \Gamma_1.input]}}$$

(a) **pipe**

$$\frac{t_{/\Gamma} \Rightarrow \sigma \star b_{/\Gamma'}}{\textbf{process}\ t_{/\Gamma} \Rightarrow \sigma \star \overline{b}_{/\Gamma[fs \leftarrow \Gamma'.fs,\ input \leftarrow \Gamma'.input]}}$$

(b) **process**

FIGURE 8 – Sémantique de l'évaluation de **pipe** et **process**

On a également le **call** : l'appel de procédure. On peut noter qu'il ne travaille pas sur un nom de procédure et des arguments, mais sur une liste dont le premier élément sera considéré comme le nom de la procédure et le reste comme les arguments. Cela fait la différence dans le cas de la liste vide ; l'appel est alors un succès. Cela fait également la différence dans certains cas de constructions curieuses de listes. Des exemples sont donnés en figures 1b et 1c.

Le **pipe** – en shell, | – permet de prendre la sortie standard d'un terme et de l'utiliser comme entrée d'un autre. On peut noter qu'il ignore totalement le comportement du premier terme. Sa sémantique se trouve en figure 8a.

Le **process**, enfin, protège une partie du contexte des modifications. Ainsi, les modifications des fonctions, des variables et des arguments à l'intérieur d'un **process** n'ont aucun impact sur l'extérieur. Les modifications concernant le système de fichiers et l'entrée standard sont – elles – conservées. Sa sémantique se trouve en figure 8b.

3. Formalisation en Why3

Dans cette section, nous décrivons la formalisation en Why3 de la définition du langage CoLiS et l'utilisation de cette définition pour assurer la correction d'un interpréteur de référence pour ce langage.

Why3 [3, 4] est un environnement de vérification déductive de programmes. Il fournit un langage de spécification et de programmation. Les théorèmes et les programmes annotés – en fait, tout ce qui demande à être prouvé – sont convertis en obligations de preuves ensuite transmises à des prouveurs externes. Son langage de programmation – WhyML – est un langage de la famille ML contenant des éléments fondamentalement impératifs comme les références ou les exceptions. Ces éléments sont finement gérés dans les obligations de preuve et permettent d'écrire naturellement des programmes sans compliquer pour autant le travail des prouveurs.

La retranscription de la syntaxe et de la sémantique est quasiment immédiate. En témoignent les figures 9, 10 et 11 qui présentent la syntaxe, les jugements sémantiques et la sémantique de l'évaluation des expressions telles que décrites en Why3. Les jugements sémantiques y sont décrits sous la forme de prédicats inductifs. Il est intéressant de mettre ces figures en parallèle avec les figures 3, 4 et 6.

Il y a tout de même quelques modifications, essentiellement pour expliciter certaines choses

exit in this case, a non-zero exit status shall be returned. Otherwise, zero shall be returned."

```
type svar = string
type lvar = string

type term =
  | TTrue
  | TFalse
  | TFatal
  | TReturn term
  | TExit term
  | TAsString svar sexpr
  | TAsList lvar lexpr
  | TSeq term term
  | TIf term term term
  | TFor svar lexpr term
  | TDoWhile term term
  | TProcess term
  | TCall lexpr
  | TShift
  | TPipe term term

with sexpr = list sfrag

with sfrag =
  | SLiteral string
  | SVar svar
  | SArg int
  | SProcess term

with lexpr = list lfrag

with lfrag =
  | LSingleton sexpr
  | LSplit sexpr
  | LVar lvar

type program =
  { p_sdecl  : list svar ;
    p_ldecl  : list lvar ;
    p_pdecls : list (string, term) ;
    p_term   : term }
```

FIGURE 9 – Syntaxe de CoLiS en Why3

```
type behaviour =
  | BNormal bool
  | BFatal
  | BReturn bool
  | BExit bool

type senv = svar → string
type lenv = lvar → list string
type penv = string → option term

type context = {
  c_fs    : filesystem  ;
  c_senv  : senv        ;
  c_lenv  : lenv        ;
  c_args  : list string ;
  c_input : string      ;
  c_penv  : penv        ;
}

inductive eval_term           term           context  string behaviour context
with eval_term_tcall      (list string)      context  string behaviour context
with eval_term_tfor  svar (list string) term context  string behaviour context

with eval_sexpr  sexpr context    string     (option bool) context
with eval_sfrag  sfrag context    string     (option bool) context

with eval_lexpr  lexpr context  (list string) (option bool) context
with eval_lfrag  lfrag context  (list string) (option bool) context
```

FIGURE 10 – Sémantique de CoLiS en Why3

passées sous silence sur le papier.

Quelques propriétés sur le langage ont été prouvées. On prouve par exemple que les sucres syntaxiques correspondant à **!**, `&&` et `||` ont bien la sémantique attendue. Sont prouvées également l'associativité de la sémantique du **pipe** et l'idempotence de celle du **process**. L'associativité de la sémantique du **pipe** stipule que, quels que soient t_1, t_2, t_3, Γ, σ, b et Γ', on a : $\textbf{pipe}\ (\textbf{pipe}\ t_1\ \textbf{into}\ t_2)\ \textbf{into}\ t_{3/\Gamma} \Rightarrow \sigma \star b_{/\Gamma'}$ si et seulement si $\textbf{pipe}\ t_1\ \textbf{into}\ (\textbf{pipe}\ t_2\ \textbf{into}\ t_3)_{/\Gamma} \Rightarrow \sigma \star b_{/\Gamma'}$. L'idempotence de la sémantique du **process**, pour sa part, signifie que, quels que soient t, Γ, σ, b et Γ', on a : $\textbf{process}\ (\textbf{process}\ t)_{/\Gamma} \Rightarrow \sigma \star b_{/\Gamma'}$ si et seulement si $\textbf{process}\ t_{/\Gamma} \Rightarrow \sigma \star b_{/\Gamma'}$.

L'interpréteur de CoLiS – développé en WhyML – est un ensemble de fonctions mutuellement récursives annotées pour certifier qu'elles se comportent comme les jugements sémantiques présentés en section 2. Ces fonctions sont écrites dans un style habituel qui combine des traits fonctionnels et impératifs. En particulier :

- Les comportements Fatal, Return b et Exit b sont gérés par des exceptions du même nom. Pour les fonctions qui peuvent lever ces exceptions, on ajoute une annotation qui y correspond. Cela n'ajoute pas de difficulté à la preuve.
- La sortie standard est une référence. Cela permet de se rapprocher plus d'un interpréteur qui affiche les résultats au fur et à mesure qu'il les produit. Cela rend cependant la preuve plus difficile, puisqu'elle force des post-conditions de la forme :

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

  pour une sortie (b, Γ') de l'interpréteur, alors que les quantifications existentielles restent compliquées pour les *SMT solvers*.
- La composition des comportements d'expressions est faite à l'aide d'une fonction auxiliaire avec un accumulateur : au lieu de récupérer les comportements sous forme d'options de ses appels récursifs pour ensuite les composer – ce que fait la sémantique –, on transmet aux appels récursifs le comportement actuel, en leur laissant le soin de le mettre à jour ou de le transmettre. Cela diminue cependant la clarté des annotations avec des post-conditions de la forme :

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

  pour une sortie (σ, b, Γ') de l'interpréteur d'expressions.

Les annotations et l'environnement de preuve Why3 nous permettent de prouver le théorème suivant :

Théorème 1 (Correction partielle de l'interpréteur)**.** *Pour toute entrée de l'interpréteur contenant un terme t, un contexte Γ et une référence contenant la* string *Ξ, si son exécution termine et renvoie (b, Γ') et change la* string *de la référence à Ξ' ; Alors il existe une* string *σ telle que $\Xi' = \Xi \cdot \sigma$, et :*

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

C'est un très bon exemple du fait qu'il est possible de prouver des propriétés non triviales comme la correction de l'interpréteur en Why3 sur ce langage.

```
| EvalSE_String : ∀ σ Γ.
  eval_sfrag_opt (SLiteral σ) Γ σ None Γ

| EvalSE_Var : ∀ x_s Γ.
  eval_sfrag_opt (SVar x_s) Γ (Γ.c_senv x_s) None Γ

| EvalSE_Arg : ∀ n Γ.
  eval_sfrag_opt (SArg n) Γ
    (match nth n Γ.c_args with None → empty_string | Some σ → σ end)
    None Γ

| EvalSE_Process : ∀ t Γ σ b Γ'.
  eval_term t Γ σ b Γ' →
  eval_sfrag_opt (SProcess t) Γ σ
    (Some (match b with BNormal True | BReturn True
                      | BExit True → True | _ → False end))
    {Γ with c_fs = Γ'.c_fs ; c_input = Γ'.c_input}

| EvalSE_Nil : ∀ Γ.
  eval_sexpr_opt Nil Γ empty_string None Γ

| EvalSE_Cons : ∀ f_s Γ σ β Γ' s σ' β' Γ''.
  eval_sfrag_opt f_s Γ σ β Γ' → eval_sexpr_opt s Γ' σ' β' Γ'' →
  eval_sexpr_opt (Cons f_s s) Γ (concat σ σ') (bocompose β β') Γ''


| EvalLF_Singleton : ∀ s Γ σ β Γ'.
  eval_sexpr_opt s Γ σ β Γ' →
  eval_lfrag_opt (LSingleton s) Γ (Cons σ Nil) β Γ'

| EvalLF_Split : ∀ s Γ σ β Γ'.
  eval_sexpr_opt s Γ σ β Γ' →
  eval_lfrag_opt (LSplit s) Γ (split σ) β Γ'

| EvalLF_Var : ∀ x_l Γ.
  eval_lfrag_opt (LVar x_l) Γ (Γ.c_lenv x_l) None Γ

| EvalLE_Nil : ∀ Γ.
  eval_lexpr_opt Nil Γ Nil None Γ

| EvalLE_Cons : ∀ f_l l Γ Γ' Γ'' λ λ' β β'.
  eval_lfrag_opt f_l Γ λ β Γ' → eval_lexpr_opt l Γ' λ' β' Γ'' →
  eval_lexpr_opt (Cons f_l l) Γ (λ ++ λ') (bocompose β β') Γ''
```

FIGURE 11 – Sémantique de l'évaluation des *string* et *list expressions* en Why3

3.1. Extraction de l'interpréteur

La preuve de l'interpréteur est composée de 146 obligations de preuve qu'il est possible de rejouer. Les sources et les instructions sont disponibles en ligne [5]. Il est suffisant d'avoir Why3 [16] (0.87.2) et les prouveurs Alt-Ergo [17] (1.01), E [18] (1.9.1) et Z3 [19] (4.4.1). Pour renforcer la preuve, les prouveurs CVC3 [20] (2.4.1), CVC4 [21] (1.4) et SPASS [22] (3.9) peuvent également être utilisés.

Une fois l'interpréteur prouvé, on peut vouloir le voir fonctionner. Pour cela, Why3 possède un mécanisme d'extraction, c'est-à-dire un mécanisme lui permettant de produire, à partir de code WhyML, du code dans un autre langage de programmation. En ce qui nous concerne, cela nous permet de produire un peu moins de 550 lignes de code OCaml [23] à partir de notre interpréteur et nous l'avons testé sur des exemples dont ceux de la figure 2. Dans l'archive [5] se trouve le code nécessaire à l'extraction vers OCaml et au lancement des tests sur les exemples donnés dans cet article.

4. Travaux connexes

Plusieurs autres travaux cherchent à donner des garanties sur les scripts shell. Par exemple, Ntzik et Gardner [6] s'attaquent au système de fichiers et à ses primitives en les spécifiant à l'aide d'une logique de séparation. Un petit langage y est également défini qui leur sert à définir des commandes dérivées sur le système de fichiers à partir de commandes plus simples – `rm -r` à partir de `rm` par exemple. Il leur a permis de repérer des erreurs dans des implémentations de `rm -r`. Ce petit langage reste très éloigné du shell, ce qui fait que l'aspect de modélisation des scripts n'est pas du tout présent dans leur travail.

D'autres travaux comme [7] et [8] se focalisent sur les langages de script et cherchent à détecter statiquement des erreurs. Dans [7], ils présentent un outil appelé Abash qui simule une exécution de scripts bash pour repérer des situations où l'expansion pourrait avoir des comportements dangereux. Cet outil leur a permis de repérer divers bugs dans un corpus de scripts. Bien que [8] soit focalisé sur PHP, de nombreuses problématiques communes aux langages y sont présentes. Ces deux papiers s'attaquent cependant directement au langage de script ciblé, ce qui limite les types de bugs qu'ils peuvent détecter.

Il n'existe, semble-t-il, pas d'autres travaux tentant de traduire d'abord un langage de script vers un langage plus propre, sur lequel il sera ensuite plus facile de prouver des propriétés.

16. `http://why3.lri.fr/`
17. `https://alt-ergo.ocamlpro.com/`
18. `http://wwwlehre.dhbw-stuttgart.de/~sschulz/E/E.html`
19. `https://github.com/Z3Prover/z3/wiki`
20. `http://www.cs.nyu.edu/acsys/cvc3/index.html`
21. `http://cvc4.cs.nyu.edu/web/`
22. `http://www.mpi-inf.mpg.de/departments/automation-of-logic/software/spass-workbench/classic-spass-theorem-prover/`
23. Dont 150 de librairie standard de Why3. L'importante différence avec les 800 lignes de code Why3 vient du prédicat pour la sémantique qui n'est pas extrait.

5. Conclusion

Nous avons défini formellement un langage de script. Ce langage élimine certains défauts du shell, et tend à rendre plus visible son fonctionnement afin d'en éviter les pièges. Nous avons mis en évidence le fait qu'il est possible d'utiliser l'environnement de preuve Why3 pour automatiser des preuves sur ce langage.

Les travaux futurs concernent la définition du système de fichiers – laissé abstrait jusqu'ici pour se focaliser sur les structures du langage. Il devra être accompagné d'une logique de spécification des propriétés à vérifier sur le système de fichiers.

À terme, nous souhaitons définir et développer l'outil de traduction de shell vers CoLiS. Cet outil se devra d'analyser statiquement les scripts shell pour certaines parties de la traduction, notamment sur l'usage des variables comme des *strings* ou listes.

Références

[1] The Debian Policy Mailing List. *Debian Policy Manual*. `https://www.debian.org/doc/debian-policy/`.

[2] IEEE and The Open Group. *POSIX.1-2008/Cor 1-2013*. `http://pubs.opengroup.org/onlinepubs/9699919799/`.

[3] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3 : Shepherd your herd of provers. In *Boogie 2011 : First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[4] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[5] Nicolas Jeannerod. Le coquillage dans le colis-mateur. `https://nicolas.jeannerod.fr/research/le-coquillage-dans-le-colis-mateur`.

[6] Gian Ntzik and Philippa Gardner. Reasoning about the POSIX file system : local update and global pathnames. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 201–220. ACM, 2015.

[7] Karl Mazurak. Abash : Finding bugs in bash scripts. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS*, 2007.

[8] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In Angelos D. Keromytis, editor, *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*. USENIX Association, 2006.