Le langage CoLiS : syntaxe, sémantique et outillage

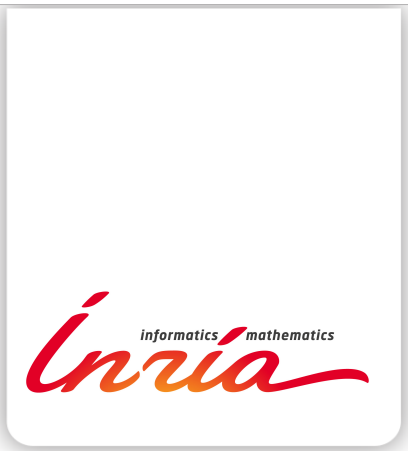
Ilham Dami, Claude Marché

Le langage CoLiS : syntaxe, sémantique et outillage

Ilham Dami, Claude Marché

TECHNICAL REPORT
N° 0491
October 2017
Project-Team Toccata

ISSN 0249-0803 ISRN INRIA/RT--0491--FR+ENG



Le langage CoLiS : syntaxe, sémantique et outillage *

Ilham Dami †, Claude Marché ‡

Équipe-Projet Toccata

Rapport technique n° 0491 — October 2017 — 22 pages

Résumé : Ce document présente les choix de conception du langage CoLiS, conçu dans le cadre du projet ANR du même nom. On présente la syntaxe du langage, à la fois sous une forme abstraite et une forme concrète, puis des conditions de bon typage et enfin sa sémantique opérationnelle. On décrit ensuite un outil qui implémente différentes opérations applicables sur les programmes CoLiS, principalement un interpréteur et un traducteur vers le langage shell. La sémantique opérationnelle est définie formellement au sein de l'environnement Why3, avec lequel une preuve formelle de correction de l'interpréteur est réalisée. Enfin, on présente un outillage visant à l'exécution de campagnes de test de non-régression et de détection d'incohérences entre l'interprétation directe et la traduction vers le shell.

Mots-clés : Interpréteur de commandes Unix, scripts shell, sémantique opérationnelle formelle

* Work supported by the CoLiS project (ANR-15-CE25-0001, http://colis.irif.fr)
† Université Paris-Sud, Orsay, France
‡ Inria & LRI, CNRS, Université Paris-Sud, Université Paris-Saclay, Orsay, France

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

The CoLiS language: syntax, semantics and associated tools

Abstract: This report presents the design choices of the CoLiS language, conceived within the ANR project of the same name. We present the syntax of the language, both in an abstract and in a concrete form, then we give typing rules, and finally its operational semantics. We describe a tool that implements various operations applicable on CoLiS programs, mainly an interpreter and a translator to the shell language. Operational semantics is defined formally within Why3, with which a formal proof of correction of the interpreter is performed. Finally, we present some tooling for the execution of test campaigns for non-regression and the detection of inconsistencies between direct interpretation and translation to the shell.

Key-words: Unix command interpreter, shell scripts, formal operational semantics

Table des matières

1 Contexte et motivations

Le *shell* désigne l'interpréteur de commandes des systèmes d'exploitation de type Unix. Les *scripts shell* sont des programmes qui contiennent une série de commandes interprétées par un shell Unix. Ils sont très utilisés lorsque l'on souhaite faire des manipulations sur le système de fichiers, et ont un rôle crucial lors de l'installation de nouveaux logiciels, auquel cas on parlera de *scripts d'installation*. Étant donné que de tels scripts sont exécutés avec les privilèges du compte `root`, qui n'a aucune restriction d'action sur le système de fichiers, une erreur dans un script d'installation peut corrompre le système de fichiers courant et rendre le système d'exploitation inutilisable.

Certains logiciels ont été développés pour tester la cohérence intrinsèque d'un script shell [1] mais ils ne cherchent pas à vérifier ses actions, et donc sa cohérence avec des spécifications externes. Or, lorsque l'on s'intéresse aux scripts d'installation, vérifier qu'ils ne cassent pas l'OS et, plus généralement, contrôler leurs interactions avec le système de fichiers est central. C'est ce que le projet ANR CoLiS [2]. vise à vérifier grâce à des techniques de vérification déductive de programmes et de transducteurs d'arbres.

Le projet CoLiS se concentre sur les scripts d'installation présents dans les paquets logiciels de la distribution Debian GNU/Linux. Afin de diriger la recherche d'erreurs dans ces scripts, nous nous baserons sur les recommandations de la *Debian Policy* [3], initialement destinée aux personnes développant pour la distribution Debian. Il existe déjà un outil permettant de tester les paquets Debian : Lintian [4]. Cependant, contrairement au projet CoLiS, cet outil se concentre sur les violations de la Debian Policy et les erreurs communes.

Bien qu'ambitieux, le but du projet CoLiS est d'être plus exhaustif dans la quête d'erreurs en ayant une vision plus large de l'interaction des scripts sur le système de fichiers, et cela, en adoptant une approche plus systématique. En particulier, nous souhaiterons que notre script soit conforme au standard POSIX et donc qu'il n'utilise aucune fonctionnalité extérieure à ce sous-ensemble. Plus généralement, nous souhaitons éviter des incohérences dans le système de fichiers ainsi que sa corruption, et ce, quelque soit l'état courant et l'activité du système, notamment lorsque la personne utilisatrice décide d'interrompre l'installation.

Pour faciliter les analyses des scripts d'installation, le projet CoLiS a défini un langage (nommé également CoLiS) qui évite des difficultés importantes et même des pièges existants au niveau de la syntaxe et de la sémantique des scripts shell [2]. Nous avons défini ce langage selon les principes contemporains du monde académique : définition d'une syntaxe *abstraite* des constructions du langage, définition du syntaxe concrète qui associe à chaque texte de programme un arbre de syntaxe abstraite, définition formelle de la sémantique opérationnelle à l'aide de règles de déduction. De plus, la syntaxe et la sémantique opérationnelle sont formalisées à l'aide de l'environnement Why3 [1]. Un interpréteur de programmes CoLiS est implémenté en WhyML et sa correction, vis-à-vis de la sémantique opérationnelle, est prouvée formellement avec Why3.

Ce rapport présente l'état courant du design du langage CoLiS (syntaxe, sémantique et interprétation) ainsi que l'outil logiciel `colis`. Cet outil a été réalisé par Ilham Dami lors de son stage au sein du projet, pour travailler de manière pratique avec les programmes écrits en CoLiS. La section 2 commence par introduire la syntaxe du langage CoLiS, puis la section 3 présente l'outil `colis` d'un point de vue de l'utilisateur. La section 4 est consacrée à sa sémantique, mise à jour par rapport aux versions précédentes [2, 3]. La section 5 présente des conditions de bon typage pour les programmes CoLiS. La section 6 est consacrée à l'interpréteur CoLiS, qui a été également mis à jour. Un travail central dans le stage était la réalisation d'un traducteur automatique de programmes CoLiS vers des scripts shell, ce

1. `https://www.shellcheck.net`, Licence GPLv3
`https://sourceforge.net/projects/shunit/`, Licence GPLv2
2. `http://colis.irif.fr`
3. `https://www.debian.org/doc/debian-policy/`
4. Lien vers Lintian : `https://lintian.debian.org`, Licence GPL(v2 ou plus récente)
Lien sur des techniques de contrôle d'erreurs : `https://www.debian.org/doc/manuals/maint-guide/checkit.fr.html`

Variables : *strings*	x_s	$\in$	$SVar$
Variables : listes	x_l	$\in$	$LVar$
Noms de procédures	c	$\in$	$\mathscr{F}$
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 | **while** t **do** t
		|	**process** t | **pipe** t **into** t
		|	**call** l | **shift**

FIGURE 1 – Syntaxe abstraite de CoLiS

travail est présenté dans la section 7. Enfin, la section 8 présente l'outillage pour réaliser des jeux de tests, et présente les résultats de tels tests.

2 Syntaxes abstraite et concrète de CoLiS

La syntaxe abstraite présentée ci-dessous est quasiment identique aux version présentées dans les travaux précédents [2, 3]. La seule différence concerne l'ajout de l'instruction de boucle `while ... do ..` . La syntaxe concrète présentée ensuite est entièrement nouvelle, puisqu'aucune telle syntaxe n'avait été présentée jusqu'à présent.

2.1 Syntaxe abstraite

La syntaxe abstraite sous forme de grammaire est donnée dans la figure 1. Une telle définition se code directement en syntaxe Why3, comme indiqué sur la figure 2. Ce code se trouve dans le fichier `model/colis/why3/base.mlw` du dépôt git CoLiS.

2.2 Syntaxe concrète

Jusqu'à présent, le langage CoLiS ne se présentait que sous la forme d'une syntaxe abstraite. Nous avons définie une syntaxe concrète, qui est présentée dans la figure 3. Les identificateurs de variables et de fonctions autorisés sont des séquences non vides de caractères alphanumériques ou de underscore ("_"), commençant par une lettre. La distinction entre majuscules et minuscules est significative. Les littéraux

```
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
  | TWhile 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
```

FIGURE 2 – Syntaxe Abstraite sous forme de code Why3

string commencent et se terminent par le caractère simple quote (“’”). un backslash (“\”) à l’intérieur d’une string échappe le caractère qui suit, en particulier la simple quote et le backslash. Attention les séquences d’échappements standards en shell comme `\n` et `\t` sont donc interprétées respectivement comme `n` et `t`, il faut écrire `'\\n'` pour obtenir la chaîne à deux caractères `'\n'`.

À noter que les fichiers sources CoLiS peuvent également être annotés par des commentaires, sous la même forme qu’en OCaml c’est-à-dire `(* <commentaire> *)`. Les commentaires peuvent être imbriqués.

La correspondance entre les règles de syntaxe concrète et la syntaxe abstraite est immédiate dans la majorité des cas, les cas moins triviaux sont énumérés ci-dessous.

— `if` t_1 `then` t_2 `fi` se traduit en syntaxe abstraite par **if** t_1 **then** t_2 **else false**.

— `not` t se traduit en **if** t **then false else true** x

— dans les règles pour $lfrag$, le membre droit $sfraglist$ correspond au singleton $[sfraglist]$ et le membre droit `split` $sfraglist$ correspond à **split** $sfraglist$

— `embed` t correspond à t dans un sfrag (`SProcess` en Why3)

Exemples Des exemples de fichiers source CoLiS se trouvent dans le répertoire `model/colis/ocaml/tests` du dépôt Git CoLiS. La figure 4 donne un exemple illustratif d’un code source CoLiS pour un exemple jouet.

Programmes	p	::=	$decl^*\ term$
Déclarations	$vdecl$	::=	`var` $id\ (\texttt{,}\ id)^*$ `: string`
		|	`var` $id\ (\texttt{,}\ id)^*$ `: list`
		|	`function` id `=` $term$
Termes	$term$	::=	`true` | `false` | `fatal`
		|	`return` $term$ | `exit` $term$
String assignment		|	id `:=` $sfraglist$
List assignment		|	id `::=` $lfraglist$
		|	`begin` seq `end`
		|	`if` $term$ `then` $term$ `else` $term$ `fi`
		|	`if` $term$ `then` $term$ `fi`
		|	`not` $term$
		|	`for` id `in` l `do` $term$ `done`
		|	`do` $term$ `while` $term$ `done`
		|	`while` $term$ `do` $term$ `done`
		|	`process` $term$
		|	`pipe` $term\ (\texttt{into}\ term)^*$ `epip`
		|	`call` $lfraglist$
		|	`shift`
	$sfraglist$	::=	`{` $sfrag\ (\texttt{,}\ sfrag)^*$ `}`
	$lfraglist$	::=	`[` $lfrag\ (\texttt{;}\ lfrag)^*$ `]`
String fragments	$sfrag$	::=	*string-literal* | id | `arg` *number* | `embed` $term$
List fragments	$lfrag$	::=	$sfraglist$ | `split` $sfraglist$ | id
	seq	::=	$term$ `;` seq

FIGURE 3 – Syntaxe concrète de CoLiS

3 Fonctionnalités de l'outil `colis`

Pour pouvoir faciliter les expérimentations et les tests sur les programmes CoLiS, un outil en ligne de commande a été conçu dans le but d'automatiser certaines tâches. Cette commande se nomme `colis` et on l'invoque de l'une des deux manières suivantes :

```
colis [options] <action> (<file.cls> | -dir <répertoire>)
colis [options] compare (<file1> <file2> | -dir <répertoire>) [-root <rfs>]
```

La suite de cette section décrit les différentes actions possibles et les différentes options qui peuvent être utilisées.

3.1 Extensions de fichier utilisées

`cls` Programme écrit en syntaxe concrète CoLiS.

`sh` Script shell.

`oracle` Fichier indiquant le comportement et la sortie attendus d'un script. Le comportement (`true` ou `false`) est indiqué sur la première ligne. La sortie est indiquée à partir de la deuxième ligne (ne tient pas compte de la sortie d'erreur). Chaque caractère étant interprété, on fera particulièrement attention à ne pas inclure de sauts de ligne superflus en fin de fichier.

`pp` Résultat du parsing d'un fichier `.cls`

```
(** [inferieur t u] renvoie [true] quand [t <= u] en interprétant les
    contenus de [t] et [u] comme des nombres *)
function inferieur = call [{'test'} ; {arg 0} ; {'-le'} ; {arg 1}]

var heure : string

(** [heure_manger] renvoie [true] s'il est entre 11 et 13 heures (inclus).
    l'heure courante est affectée à la variable globale [heure] *)
function heure_manger =
  begin
    heure := {embed call [{'date'}; {'+%H'}]} ;
    if call [{'inferieur'} ; {heure} ; {'10'}]
    then false
    else call [{'inferieur'} ; {heure} ; {'13'}]
    fi
  end

var menu : list

(** [dans_menu t] renvoie [true] si la chaine [u] apparait dans la
    liste globale [menu] *)
function dans_menu =
  begin
    for i in [menu] do
      if call [{'test'}; {i}; {'='}; {arg 0}] then
         return true
      fi
    done ;
    return false
  end

(** programme principal, quelques tests jouets *)
begin
  menu ::= [split {'quinoa'} ; split {'poisson'} ; split {'pomme-de-terre'} ;
           split {'jus_d\'orange'}] ;
  if call [{'dans_menu'} ; {'quinoa'}] then
    if call [{'dans_menu'} ; {'jus_de_pomme'}] then
      call [{'echo'} ; {'Tout ce que j\'aime! (debug)'}]
    else
      if call [{'dans_menu'} ; {'jus_d\'orange'}] then
        if call [{'dans_menu'} ; {'pomme-de-terre'}]
        then call [{'echo'} ;
               {'Il n\'y a pas ma boisson préférée... Tant pis!'}]
        else call [{'echo'} ; {'Bof... (debug)'}]
        fi
      fi
    fi
  fi ;
  if call [{'heure_manger'}] then
    call [{'echo'} ; {'C\'est l\'heure d\'aller manger!'}]
  else
    call [{'echo'} ; {'Ce n\'est pas l\'heure d\'aller manger...'}]
  fi ;
  exit true
end
```

FIGURE 4 – Fichier exemple tests/quinoa.cls

ml Fichier source OCaml

3.2 Actions

ast Génère l'arbre (ou les arbres) de syntaxe abstraite CoLiS correspondant(s) au fichier `.cls`. Le fichier résultant est un fichier de même nom mais d'extension `.ml`, contenant une variable `prog` de type `program`. Aucune autre information n'est présente dans le fichier, comme les dépendances, et n'est donc pas compilable en l'état.

pp Parse un programme CoLiS et le réimprime avec un pretty-printer codé en OCaml. Si le parsing se déroule sans erreur, l'impression du résultat est identique au fichier `.cls` d'entrée, modulo les sauts de ligne et l'indentation. Cela sert donc simplement à tester le parser.

typing Vérifie le bon typage d'un programme CoLiS.

exec Exécute un programme CoLiS avec l'interpréteur CoLiS et affiche le résultat sur la sortie standard.

oracle Exécute un programme CoLiS avec l'interpréteur CoLiS et écrit le résultat dans un fichier de même nom et d'extension `.oracle`

sh Convertit un programme CoLiS en un script shell équivalent. Le résultat est écrit dans un fichier de même nom et d'extension `.sh`

clean Supprime tous les fichiers du répertoire spécifié avec l'option `-dir` exceptés les fichiers `.cls` et `.oracle`.

compare Compare le comportement (`true` ou `false`) et la sortie des deux fichiers spécifiés en argument. Le comportement et la sortie d'un fichier `.cls` ou `.pp` sont ceux de son exécution avec l'interpréteur CoLiS, ceux d'un fichier `.sh` sont ceux de son exécution avec le shell `/bin/sh` en mode strict (pour cela il faut obligatoirement spécifier via l'option `root` la racine d'un système de fichier dans lequel sera exécuté le script). Quant au fichier `.oracle`, il indique le comportement attendu sur sa première ligne, et la sortie attendue à partir de la deuxième ligne jusqu'à la fin du fichier : aucun saut de ligne superflu n'est toléré.

3.3 Options

-debug Affiche quelques informations basiques sur la commande entrée avec ses paramètres et éventuellement quelques informations sur l'avancement de son exécution.

-dest Spécifie le répertoire dans lequel on souhaite enregistrer le(s) fichier(s) généré(s).

-dir Spécifie le répertoire dans lequel se trouvent les fichiers à traiter dans le cas où l'on souhaite lancer une action sur plusieurs fichiers d'un coup. Cette option est utilisable avec n'importe quel type d'action. L'action `compare` comparera alors tous les fichiers avec leur `.oracle` associé (même nom, extension différente) : si un fichier du répertoire spécifié n'a pas de `.oracle` associé dans ce même répertoire, il sera ignoré.

-f Force l'exécution en répondant 'oui' à toutes les questions posées.

-fspath Indique que le chemin des fichiers spécifiés en argument est relatif au système de fichier spécifié avec `-root`.

-optim Optimise la syntaxe du script shell utilisé (pertinent qu'avec l'action `sh`). Le code obtenu est plus lisible.

-s Pour utiliser `schroot` au lieu de `chroot`.

-stdout Affiche le résultat de l'exécution de la commande sur la sortie standard.

-root Spécifie la racine du système de fichier dans lequel sera exécuté un script shell pour pouvoir récupérer son comportement et sa sortie. Cette option requiert que `chroot` soit installé sur la machine utilisée (utiliser `-s` pour utiliser `schroot` à la place).

3.4 Préparation d'un système de fichier auxiliaire

L'exécution des scripts, que soit par l'interpréteur CoLiS ou par traduction vers sh, peut potentiellement exécuter n'importe quelle commande de base, par exemple `rm`. Pour éviter tout risque d'impact fâcheux sur le système de fichier en cours, ces exécutions doivent se faire dans un système de fichier auxiliaire, dont le chemin est donné par l'option `-root`. Comme un changement de racine doit être effectué, il est nécessaire de lancer la commande colis en tant qu'utilisateur `root`, Même si cela semble dangereux, cela revient à faire confiance uniquement au programme `colis`, alors que sans cela il faudrait faire confiance à n'importe quel script `.cls` d'origine quelconque.

Une manière simple de préparer un système de fichier auxiliaire adéquat est d'utiliser la commande `debootstrap` de Debian. Voici par exemple comment créer une image du système de fichier de base dans le répertoire `myroot` de son home directory.

```
sudo debootstrap stable ~/myroot
```

3.5 Exemple

En utilisant le fichier `quinoa.cls` présenté dans la section précédente, on peut réaliser les opérations suivantes :

— vérifier la syntaxe et réimprimer

```
> colis pp tests/quinoa.cls
file created: tests/quinoa.pp
```

Remarquer que le texte du programme dans le fichier `.pp` est remanié : en particulier les déclarations de variables sont regroupées en tête de fichier.

— vérifier le typage

```
> colis typing tests/quinoa.cls
Well-typed program
```

Noter que naturellement, en cas d'erreur de syntaxe ou de typage, l'erreur et sa localisation dans le fichier est indiquée

— Exécuter avec l'interpréteur CoLiS

```
> sudo colis exec -root ~/myroot tests/quinoa.cls
STDOUT: Il n'y a pas ma boisson préférée... Tant pis!
Ce n'est pas l'heure d'aller manger...

BEHAVIOUR: true
```

— Traduire vers un script shell

```
> colis sh tests/quinoa.cls
file created: tests/quinoa.sh
```

L'exécution du script obtenu peut naturellement être lancée avec `sh tests/quinoa.sh` mais attention aux risques d'impact sur votre propre système de fichiers ! Il est sans doute sage d'utiliser la commande qui suit

— Exécuter les scripts CoLiS et Shell et comparer les résultats

```
> sudo colis compare -root ~/myroot tests/quinoa.cls tests/quinoa.sh
Ok.
```

4 Sémantique opérationnelle

La sémantique opérationnelle de CoLiS a été décrite dans des travaux précédents [2, 3] que nous ne répétons pas ici. Cette sémantique est définie formellement par des jugements et des règles d'inférence de jugements valides. Le jugement principal concerne l'exécution des termes, il est de la forme

$$t_{/\Gamma_1} \Downarrow \sigma \star \beta_{/\Gamma_2}$$

qui signifie que dans l'état courant Γ_1, l'exécution du terme t termine dans l'etat Γ_2, en écrivant σ sur la sortie standard, et renvoyant le comportement β. L'ensemble des comportements est défini ci-dessous.

Depuis ces travaux précédents, il a été décidé de rajouter une boucle de type `while ... do ..`

4.1 Comportement de termes

On rappelle que la sémantique se base sur la notion de comportement, dont les différents cas sont décrits dans le tableau suivant.

Termes	Équivalent shell	Comportement correspondant
true	`true`	Normal `true`
false	`! true`	Normal `false`
fatal	`false`	Fatal
return true	`return true`	Return `true`
return false	`return false`	Return `false`
exit true	`exit true`	Exit `true`
exit false	`exit false`	Exit `false`

On notera en particulier la différence entre deux comportements : Normal `false` et Fatal. La Debian Policy encourage fortement l'utilisation du mode strict [5] : Fatal correspond à un code de retour non nul qui fait l'objet d'une levée d'exception alors que Normal `false` correspond à un code de retour non nul sans levée d'exception. Par exemple en shell, sans le mode strict, les exécutions de `false` d'une part et de `! true` d'autre part donnent le même code de retour (`1`). En mode strict, `false` interrompt le shell dans lequel il est exécuté alors que `! true` ne l'interrompt pas et a un code de retour qui vaut `1`. Fatal est ainsi l'équivalent en shell `false` et Normal `false` l'équivalent de `! true`.

4.2 Sémantique du while

La sémantique de CoLiS ne contenait jusqu'à présent pas de terme **while** représentant fidèlement la boucle `while` du shell. Ce dernier était plutôt représenté en CoLiS à l'aide d'un **do while** combiné à un **if** [6]. Cette solution avait été retenue car la sémantique du **do while** était plus simple à écrire que celle du **while** [2]. En effet, si on ne considère pas les cas particuliers d'un comportement anormal en mode strict (correspondant au comportement **Fatal** de CoLiS), d'un `return` ou d'un `exit` situé dans la condition ou le corps de la boucle, le code de retour d'une boucle `while` dans un script shell est soit `0` si on n'entre jamais dans la boucle (i.e. si on n'exécute jamais le corps de la boucle), soit le code de retour de l'exécution du corps de la boucle lors de sa dernière itération [7]. Donc pour écrire la sémantique du **while** de CoLiS, il

5. En mode strict (obtenu grâce à l'instruction `set -e`), une instruction ayant un code de retour non nul interrompt l'exécution du script (à quelques exceptions près).

6. L'instruction shell `while <`t_1`>; do <`t_2`>; done` était jusqu'à présent représentée en CoLiS par l'expression : **if** t_1 **then** (**do** t_2 **while** t_1) **else true**.

7. Valable si la sortie de la boucle se fait lorsque la condition de celle-ci n'est plus vérifiée, et non via un `break`, auquel cas le code de retour sera `0`. Ce dernier n'a pour le moment pas encore été modélisé en CoLiS du fait de sa faible occurrence dans les scripts shell et par les potentielles difficultés que cela introduira. On n'en tient donc pas compte ici.

$$\frac{(t_1, \texttt{true}, t_2)_{/\Gamma} \Downarrow_w \sigma_1 \star \beta_{/\Gamma_1}}{(\textbf{while}\ t_1\ \textbf{do}\ t_2)_{/\Gamma} \Downarrow \sigma_1 \star \beta_{/\Gamma_1}}\ \textsc{While}$$

$$\frac{\begin{array}{c} t_{1/\Gamma} \Downarrow \sigma_1 \star \text{Normal } \texttt{true}_{/\Gamma_1} \\ t_{2/\Gamma_1} \Downarrow \sigma_2 \star \text{Normal } b_{1/\Gamma_2} \\ (t_1, b_1, t_2)_{/\Gamma_2} \Downarrow_w \sigma_3 \star \beta_{/\Gamma_3} \end{array}}{(t_1, b, t_2)_{/\Gamma} \Downarrow_w \sigma_1\sigma_2\sigma_3 \star \beta_{/\Gamma_3}}\ \textsc{True} \qquad \frac{t_{1/\Gamma} \Downarrow \sigma_1 \star \beta_{/\Gamma_1} \qquad \beta \in \{\text{Normal } \texttt{false}, \text{Fatal}\}}{(t_1, b, t_2)_{/\Gamma} \Downarrow_w \sigma_1 \star \text{Normal } b_{/\Gamma_1}}\ \textsc{False}$$

$$\frac{\begin{array}{c} t_{1/\Gamma} \Downarrow \sigma_1 \star \text{True}_{/\Gamma_1} \\ t_{2/\Gamma_1} \Downarrow \sigma_2 \star \beta_{/\Gamma_2} \qquad \beta \in \{\text{Fatal}, \text{Return } \_, \text{Exit } \_\} \end{array}}{(t_1, b, t_2)_{/\Gamma} \Downarrow_w \sigma_1\sigma_2 \star \beta_{/\Gamma_2}}\ \textsc{Transmit-Body}$$

$$\frac{t_{1/\Gamma} \Downarrow \sigma_1 \star \beta_{/\Gamma_1} \qquad \beta \in \{\text{Return } \_, \text{Exit } \_\}}{(t_1, b, t_2)_{/\Gamma} \Downarrow_w \sigma_1 \star \beta_{/\Gamma_1}}\ \textsc{Transmit-Cond}$$

FIGURE 5 – Règles sémantiques du **while**

faut garder en mémoire le code de retour de l'itération précédente. On introduit alors le nouveau jugement suivant :

$$(t_1, b, t_2)_{/\Gamma} \Downarrow_w \sigma \star \beta_{/\Gamma'}$$

avec

$$\begin{array}{rcl} t_1, t_2 & \in & \mathit{Term} \\ b & \in & \mathit{Boolean} \\ \Gamma, \Gamma' & \in & \mathscr{F}\mathscr{S} \times \mathit{String} \times \mathit{StringList} \times \mathit{SEnv} \times \mathit{LEnv} \\ & & \text{où} \quad \mathscr{F}\mathscr{S} \text{ est un système de fichiers} \\ & & \text{et} \quad \mathit{StringList} \triangleq \{\sigma^* \mid \sigma \in \mathit{String}\} \\ & & \mathit{SEnv} \triangleq [\mathit{SVar} \rightharpoonup \mathit{String}] \\ & & \mathit{LEnv} \triangleq [\mathit{LVar} \rightharpoonup \mathit{StringList}] \\ \sigma & \in & \mathit{String} \\ \beta & \in & \{\text{Normal } \texttt{true}, \text{Normal } \texttt{false}, \text{Fatal}, \text{Return } \texttt{true}, \text{Return } \texttt{false}, \text{Exit } \texttt{true}, \text{Exit } \texttt{false}\} \end{array}$$

où Normal b correspond au comportement de l'exécution du corps de la boucle lors de sa dernière itération. La sémantique résultante du **while** est décrite dans la Figure 5.

4.3 Formalisation avec l'environnement Why3

On utilise Why3, un environnement dédié à la vérification déductive de programmes, pour l'implémentation du **while** dans la sémantique de CoLiS et garantir que l'interpréteur est correct.

La traduction des règles sémantiques du **while** (cf. Figure 5) vers Why3 est quasiment immédiate (cf. Figure 6).

```
inductive eval_term term context string behaviour context =
| ...
| EvalT_While : forall t_1 t_2 gamma sigma_1 b_1 gamma_1.
    eval_term_twhile t_1 (BNormal True) t_2 gamma sigma_1 b_1 gamma_1 →
    eval_term (TWhile t_1 t_2) gamma sigma_1 b_1 gamma_1

with eval_term_twhile term behaviour term context string behaviour context =
| EvalT_While_True : forall t_1 b_1 gamma sigma_1 gamma_1 t_2 sigma_2 b_2 gamma_2 sigma_3 b_3 gamma_3.
    eval_term t_1 gamma sigma_1 (BNormal True) gamma_1 →
    eval_term t_2 gamma_1 sigma_2 (BNormal b_2) gamma_2 →
    eval_term_twhile t_1 (BNormal b_2) t_2 gamma_2 sigma_3 b_3 gamma_3 →
    eval_term_twhile t_1 (BNormal b_1) t_2 gamma (concat (concat sigma_1 sigma_2) sigma_3) b_3 gamma_3

| EvalT_While_False : forall t_1 gamma sigma_1 b_1 gamma_1 t_2 b.
    eval_term t_1 gamma sigma_1 b_1 gamma_1 →
    (match b_1 with BNormal False | BFatal → true | _ → false end) →
    eval_term_twhile t_1 (BNormal b) t_2 gamma sigma_1 (BNormal b) gamma_1

| EvalT_While_Transmit_Body : forall t_1 gamma sigma_1 gamma_1 t_2 b_2 sigma_2 gamma_2 b.
    eval_term t_1 gamma sigma_1 (BNormal True) gamma_1 →
    eval_term t_2 gamma_1 sigma_2 b_2 gamma_2 →
    (match b_2 with BNormal _ → false | _ → true end) →
    eval_term_twhile t_1 (BNormal b) t_2 gamma (concat sigma_1 sigma_2) b_2 gamma_2

| EvalT_While_Transmit_Cond : forall t_1 gamma sigma_1 b_1 gamma_1 t_2 b.
    eval_term t_1 gamma sigma_1 b_1 gamma_1 →
    (match b_1 with BExit _ | BReturn _ → true | _ → false end) →
    eval_term_twhile t_1 (BNormal b) t_2 gamma sigma_1 b_1 gamma_1
```

FIGURE 6 – Version Why3 des règles sémantiques du `while`.

5 Typage

Pour le moment, la vérification du bon-typage d'un programme CoLiS consiste seulement à vérifier que toutes les variables utilisées dans le programme ont été déclarées au préalable, et avec le bon type (`string` ou `list`). De plus, les variables déclarées mais non utilisées sont données à titre indicatif.

On vérifie le bon typage d'un programme CoLiS en vérifiant qu'il satisfait aux deux critères suivants :

— toutes les variables utilisées dans le corps du programme et dans la définition des fonctions ont été déclarées au préalable comme une chaîne de caractères (*var <x> : string*) ou comme une liste (*var <x> : list*) ;

— l'utilisation d'une variable est cohérente avec son type (déterminé lors de sa déclaration) ;

— aucun **return** ne se trouve en dehors d'une fonction.

De plus, on signale à titre indicatif :

— la présence d'un singleton au sein de la valeur affectée à une variable de type *list*,

— la présence de variables déclarées mais non utilisées dans le programme.

6 Interpréteur CoLiS

Un interpréteur pour le langage CoLiS a été élaboré et sa sûreté et sa complétude ont été prouvées dans l'environnement Why3 lors de précédents travaux [3].

```
let rec interp_term (t: term) (g: context) (stdout : ref string) (ghost sk: skeleton) : (bool, context)
  requires { exists s b g'. eval_term t g s b g' sk }
  variant { size sk }
  returns { (b, g') →
    exists s. !stdout = concat (old !stdout) s ∧ eval_term t g s (BNormal b) g' sk }
  raises  { EFatal g' →
    exists s. !stdout = concat (old !stdout) s ∧ eval_term t g s  BFatal g' sk }
  raises  { EReturn (b, g') →
    exists s. !stdout = concat (old !stdout) s ∧ eval_term t g s (BReturn b) g' sk }
  raises  { EExit (b, g') →
    exists s. !stdout = concat (old !stdout) s ∧ eval_term t g s (BExit   b)  g' sk }
=
  match t with
  | ...
  | TWhile t_1 t_2 →
    let last_behaviour_while = ref true in
    let gamma_curr = ref g in
    let ghost sigma_init = !stdout in
    let ghost skf = skeleton1 sk in
    let ghost sk_curr = ref skf in
    while (let ghost sk1 = skeleton123 !sk_curr in
             let b_1,gamma_1 =
             try
               interp_term t_1 !gamma_curr stdout sk1
             with
               EFatal gamma_1 → (false, gamma_1)
             end
           in
           gamma_curr := gamma_1;
           b_1) do
      invariant { !sk_curr ≠ S0 }
      invariant { size !sk_curr < size sk }
      invariant { exists sigma2 bhv gamma_end.
                  eval_term_twhile t_1
                    (BNormal !last_behaviour_while) t_2 !gamma_curr sigma2 bhv gamma_end !sk_curr }
      invariant { exists sigma1. !stdout = concat sigma_init sigma1 ∧
            forall sigma2 bhv gamma_end.
            eval_term_twhile t_1
                    (BNormal !last_behaviour_while) t_2 !gamma_curr sigma2 bhv gamma_end !sk_curr →
            eval_term_twhile t_1 (BNormal True) t_2 g (concat sigma1 sigma2) bhv gamma_end skf }
      variant { size !sk_curr }
      let ghost _,sk2 = skeleton23 !sk_curr in
      let (b_2, gamma_2) = interp_term t_2 !gamma_curr stdout sk2 in
      last_behaviour_while := b_2;
      gamma_curr := gamma_2;
      let ghost _,_,sk3 = skeleton3 !sk_curr in
      sk_curr := sk3
    done;
    (!last_behaviour_while, !gamma_curr)
```

FIGURE 7 – Code Why3 de l'exécution des boucles `while`.

Après avoir modifié la sémantique de CoLiS en lui ajoutant le **while**, et complété la formalisation Why3 de cette sémantique, nous avons donc complété l'interpréteur CoLiS. Une première façon de faire est d'utiliser une fonction auxiliaire récursive pour évaluer les règles du jugement auxiliaire introduit précédemment. Une deuxième façon de faire est d'utiliser directement la boucle **while** de Why3. Ces deux méthodes ont été implémentées et ont été vérifiées mais celle retenue est la version utilisant la boucle **while** de Why3. À noter que pour prouver cette deuxième version, une difficulté inattendue est apparue, pour la preuve de terminaison qui intervient pour prouver la complétude de l'interpréteur. En effet, la version précédente utilisait une technique originale basée sus la notion de *squelette* de preuve d'un prédicat inductif. La preuve de terminaison se base sur la décroissance structurelle du squelette. Avec la version itérative du `while`, la décroissance n'est plus strictement structurelle (c'est-à-dire sur les sous-termes immédiats du squelette), c'est seulement la *taille* du squelette qui décroit. Autrement dit nous avons du passer d'une récursion primitive à une récursion générale. Une fois que cette difficulté à été identifiée et comprise, la mise à jour des preuves n'a pas posé plus de difficultés, en particulier la preuve formelle se déroule uniquement avec les prouveurs automatiques couramment disponibles dans Why3. Les détails de la preuve sont consultables dans le fichier `model/colis/why3/interpreter.mlw` du Git CoLiS, dont un extrait est montré sur la figure 7.

Le code de l'interpréteur prouvé ainsi obtenu peut être extrait vers OCaml. Ce code OCaml a été ensuite intégré à l'outil `colis`. Ce code permet alors d'exécuter les scripts CoLiS, dans le but de déterminer leur sortie ainsi que leur comportement. Une brique est néanmoins manquante dans cette approche : l'interpréteur codé en Why3 ne spécifie rien sur les commandes primitives du système d'exploitation, comme `cp`, `rm`, `mkdir`, `rmdir`, etc. Jusqu'à maintenant, cette brique manquante était remplie par un bout de code OCaml réalisant seulement les commandes `echo` et `read`. Dorénavant, les commandes sont interprétées directement par le shell, en utilisant la commande `chroot` sur la racine du système de fichiers spécifiée pour s'assurer de l'intégrité de notre système de fichier courant. Cette version de l'interpréteur n'est utilisable que via la commande `colis` pour s'assurer que la racine du système de fichier dans lequel exécuter la commande ait été spécifiée (usage :`colis exec <fichier.cls> -root <fs>`, cf. Section 3).

7 Traducteur de programmes CoLiS vers des scripts shell

Nous souhaitons pouvoir générer des scripts shell à partir de la version CoLiS du programme. Bien que cela ne soit pas pertinent dans la vérification même d'un script shell (étant donné que le projet n'a pas vocation à corriger le code mais à détecter d'éventuelles incohérences et en donner des indications), cela permettra de générer des scripts "certifiés CoLiS", ce qui offrira une meilleure visibilité du projet dans la communauté FLOSS [8] (et au-delà).

Les règles de la syntaxe abstraite ayant une correspondance précise en shell, sa traduction ne présente pas de difficulté majeure. La correspondance CoLiS vers shell est présentée dans la figure 8, où la notation <*t*> désigne la traduction de t. La fonction auxiliaire $mark\_sq(s)$ consiste à interpréter spécifiquement les simple quotes de la chaîne littérale s, comme décrit ci-dessous.

7.1 Choix entre les guillemets simples et doubles

Nous souhaitons parfois retranscrire une chaîne de caractères c telle quelle au shell, c'est-à-dire comme une chaîne dont aucun caractère n'est interprété, notamment via le phénomène d'expansion. Par exemple, {'un p\'tit exemple avec un $x'} doit être traduit en shell par `"un p\'tit exemple avec un \$x"`, en échappant le `$`, et non comme `"un p\'tit exemple avec un $x"` auquel cas "`$x`" serait remplacé par le contenu de la variable `$x`.

8. *Free/Libre and Open Source Software*, cf. https://www.gnu.org/philosophy/floss-and-foss

type Ocaml	syntaxe abstraite	shell
`term :`	TTrue	`true`
	TFalse	`! true`
	TFatal	`false`
	TReturn(t)	`<`t`> ; return $?`
	TExit(t)	`<`t`> ; exit $?`
	TAsString(s, sl)	`<`s`>=<`sl`>`
	TAsList(s, ll)	`<`s`>=<`ll`>`
	TSeq(t_1, t_2)	`<`t_1`> ; <`t_2`>`
	TIf(t_1, t_2, t_3)	`if <`t_1`> ; then <`t_2`> ; else <`t_3`> ; fi` [9]
	TFor(s, ll, t)	`for <`s`> in <`ll`> ; do <`t`> ; done`
	TDoWhile(t_1, t_2)	`while : ; do <`t_1`> ; <`t_2`> || break ; done` [10]
		`<`t_1`> ; while <`t_2`> ; do <`t_1`> ; done`
	TWhile(t_1, t_2)	`while <`t_1`> ; do <`t_2`> ; done`
	TProcess(t)	`(<`t`>)`
	TCall(ll)	`<`ll`>`
	TShift	`shift`
	TPipe(t_1, t_2)	`<`t_1`> | <`t_2`>`
`lfrag list :`	ll	$\begin{cases} \texttt{<}ll_1\texttt{>'␣'<}ll_2\texttt{>'␣'...<}ll_n\texttt{>} & \textit{si dans un TAsList} \\ \texttt{<}ll_1\texttt{>␣<}ll_2\texttt{>␣...<}ll_n\texttt{>} & \textit{sinon} \end{cases}$
`lfrag :`	LSingleton(sl)	`'<`sl`>'`
	LSplit(sl)	`<`sl`>`
	LVar(s)	`$<`s`>`
`sfrag list :`	sl	`<`sl_1`><`sl_2`>...<`sl_n`>`
`sfrag :`	SLiteral(s)	*mark_sq*(s)
	SVar(s)	$\begin{cases} \texttt{\$}s & \textit{si dans un LSplit} \\ \texttt{"\$}s\texttt{"} & \textit{sinon} \end{cases}$
	SArg(i)	$\begin{cases} \texttt{\$}(i+1) & \textit{si dans un LSplit} \\ \texttt{"\$}(i+1)\texttt{"} & \textit{sinon} \end{cases}$
	SProcess(t)	`$(<`t`>)`

FIGURE 8 – Traduction de la syntaxe abstraite CoLiS vers shell.

Pour cela, les guillemets simples `'` sont plus adéquats que les guillemets doubles car leur particularité est de n'interpréter aucun caractère autre que le guillemet simple (quelques particularités sont à noter toutefois selon le shell utilisé). L'utilisation des guillemets doubles aurait nécessité au préalable l'échappement de tous les caractères spéciaux.

Cependant, lorsque l'on souhaite utiliser un guillemet simple `'` au sein d'une chaîne de caractères, il faut le « marquer » pour qu'il soit considéré comme un élément de la chaîne de caractère. C'est le rôle de la fonction auxiliaire `mark_sq` : comme l'échappement n'est pas possible au sein de guillemets simples, on l'isole avec des guillemets doubles de la manière suivante (en reprenant l'exemple du dessus) : `'un p'"'"'tit exemple avec un $x'`.

7.2 Listes

En shell, toutes les variables sont des chaînes de caractères. Lorsque l'on souhaite utiliser une variable comme une liste, on a besoin de spécifier un caractère séparateur tel que son utilisation dans la chaîne de

caractères marque la délimitation entre les différents éléments de la liste.

Le séparateur du shell est déterminé via la variable d'environnement `$IFS`. On détermine sa valeur grâce à la commande

```
set | grep ^IFS=
```

qui retourne généralement

```
IFS=$' \t\n
```

Par défaut elle vaut donc <espace><tabulation><nouvelle ligne>, ce qui veut dire que l'utilisation d'un de ces trois caractères permet de délimiter les éléments d'une liste. Une succession de caractères séparateurs revient à un seul caractère séparateur (donc pas d'élément vide). L'inconvénient est que l'on ne peut pas spécifier un élément contenant un de ces caractères séparateurs, le caractère <espace> étant plus problématique que les autres généralement. Par exemple, si l'on souhaite parcourir une liste de noms de fichiers, il faut donc s'assurer qu'aucun nom de fichier ne comporte d'espace. Si on souhaite utiliser une variable *x* comme une liste (dans une boucle `for` par exemple), on utilisera la valeur de `$x` (et non de `"$x"` ni de `'$x'` qui n'interprètent pas les espaces).

```
> x='a b c'
> for i in $x ; do echo $i ; done
a
b
c
```

Une solution pour la prise en compte des espaces dans des éléments d'une liste est d'affecter à une variable une liste sous forme de plusieurs chaînes de caractères distincts dans des parenthèses et d'utiliser la valeur de `"${x[@]}"`.

```
> x=('a' 'b c')
> for i in "${x[@]}" ; do echo $i ; done
a
b c
```

Malheureusement, cette solution n'est conforme ni au standard POSIX, ni à la Debian Policy. Une autre solution pourrait être de modifier dynamiquement la variable d'environnement `$IFS` en fonction de la chaîne de caractères que l'on doit traiter comme un singleton (dans le cas général on cherchera à ce que cette variable ne tienne plus compte du caractère <espace>, celui-ci étant le plus problématique). Cependant, modifier cette variable est assez risquée, notamment parce que plusieurs autres programmes l'utilisent.

La mesure prise dans CoLiS pour éviter ce genre d'erreurs est d'**interdire l'interprétation de chaîne de caractères en tant que singleton dans la valeur affectée à une variable de type *list***. Autrement dit, dans toute affectation de variable de type list

$$x ::= lfraglist$$

le membre droit $lfraglist$ ne doit contenir aucun singleton, mais uniquement des variables ou des splits.

7.3 Optimisation

Le traducteur se présente sous deux versions : une version optimisée et une version non optimisée.

Programe CoLiS	Script shell correspondant traduit depuis le traducteur CoLiS/shell :	
	non optimisé	optimisé
var x : list begin x : := [split {'a b'} ; split {'c $d'} ; split {'e\'f'}] ; for i in [x] do call [{'echo'} ; {'i : ', i}] done end	`#!/bin/sh`[11] `set -e` `x='a b'' ''c $d'' ''e'"'"'f';` `for i in $x;` `do` `'echo' 'i: '"$i"''` `done`	`#!/bin/sh` `set -e` `x='a b c $d e'"'"'f';` `for i in $x;` `do` `echo 'i: '"$i"` `done`

FIGURE 9 – Exemple de traduction optimisée versus non-optimisée

if true then t_2 **else** t_3	<t_2>
if (**false** | **fatal**) **then** t_2 **else** t_3	<t_3>
if t_1 **then** (**do** t_4 **while** t_1) **else false**	`while` <t_1> `; do` <t_4> `; done`
if t_1 **then true else** t_3, $t_3 \neq t$ **;** t'	`(` <t_1> `) ||` <t_3>
if t_1 **then** t_2 **else false**, $t_2 \neq t$ **;** t'	`(` <t_1> `) &&` <t_2>
if t_1 **then** t_2 **else** t_3 *(défaut)*	`if` <t_1> `; then` <t_2> `; else` <t_3> `; fi`

FIGURE 10 – Optimisation du terme CoLiS **if** t_1 **then** t_2 **else** t_3

Optimisation syntaxique. L'optimisation concerne la syntaxe d'une part, afin de rendre le code plus clair pour une personne lisant le script. Par défaut, un singleton est traduit en shell en insérant des guillemets simples `'` autour de l'expression s'il s'agit d'une chaîne de caractères, des guillemets doubles s'il s'agit de variables. Bien que cela soit correct, la présence de tels guillemets n'est pas tout le temps indispensable et alourdit la lecture du code. Une première optimisation consiste donc à mettre des guillemets simples autour d'une chaîne de caractères à considérer comme un singleton seulement si celle-ci contient des caractères spéciaux, notamment ceux sujets au phénomène d'expansion. Les autres optimisations syntaxiques implémentées concernent aussi l'utilisation des guillemets simples dans les singletons. Par exemple, dans la version non optimisée, un guillemet simple est systématiquement ouvert quand on entre dans un singleton, systématiquement fermé quand on en sort, donc les éventuelles variables situées à l'intérieur doivent être précédées et suivis d'un guillemet simple pour en rester en-dehors. Si le singleton commence ou finit par une variable, ou contient une succession de variables, ou qu'il y a une succession de singletons, alors il y aura un surplus de guillemets simples. L'exemple de la figure 9 illustre la différence entre la traduction optimisée d'un programme CoLiS et celle non optimisée.

Toutes ces optimisations rendent le code du programme traducteur moins clair, donc plus difficile à vérifier et aussi plus propice aux bugs, d'où l'intérêt de garder une version sans aucune optimisation.

Optimisation sémantique. L'optimisation concerne aussi la sémantique, précisément lors de la traduction du terme CoLiS **if** t_1 **then** t_2 **else** t_3. La Table 10 présente les optimisations implémentées pour ce terme.

La version non optimisée utilise systématiquement la traduction par défaut.

8 Campagne de tests

Pour valider le parser réalisé et le traducteur vers shell, nous avons mis en place un script d'exécution systématique de tests, décrit ci-dessous. En plus de la commande `colis`, ce script fait appel à un autre outil réalisé dans le projet CoLiS : le parser de scripts shell **Morbig** [4]. Une analyse statistique a été réalisée sur un corpus de plus de 30 000 scripts d'installation Debian.

8.1 Script `run_tests`

Le script `run_tests.sh` permet de réaliser à la volée une série de tests sur les programmes placés dans le dossier `./tests` en utilisant entre autres la commande `colis`. Il vérifie que :

— les programmes placés dans le sous-dossier `badtyping` ne sont pas bien typés,

— les autres programmes sont bien typés,

— le comportement et la sortie de chaque programme attendus par l'interpréteur CoLiS sont les mêmes que ceux indiqués dans le fichier **.oracle** correspondant,

— génère les scripts shell avec le traducteur non optimisé d'une part et optimisé d'autre part, les parse avec **Morbig** et compare leur exécution avec ce qui est indiqué dans les fichiers **.oracle** correspondants.

On lance la série de tests avec :

```
sudo MORBIG=<path of>/morbig sh run_tests.sh <fs root>/
```

où `<fs root>` est le nom de la racine du système de fichiers dans lequel on souhaite que les scripts shell générés soient exécutés pour les tests. Le script `run_tests.sh` utilise la commande `chroot`.

Après avoir vérifié empiriquement l'implémentation du traducteur CoLiS/shell, du parser et du test de typage d'un programme CoLiS (écrit en syntaxe concrète), on vérifie la cohérence entre l'interprétation des programmes CoLiS par l'interpréteur CoLiS et l'exécution shell des scripts correspondants obtenus grâce au traducteur CoLiS/shell. Pour cela, on compare la sortie d'un programme CoLiS donnée par l'interpréteur avec la **sortie standard** [12] du script shell correspondant, ainsi que leur comportement. Le comportement, un booléen, est vrai – **true** – si le code du retour du script shell est nul, faux – **false** – sinon. En cas d'incohérence, il s'agira de savoir si le problème vient de l'interpréteur CoLiS ou du traducteur CoLiS/shell.

8.2 Jeux de tests

Les tests effectués sont ceux présents dans le répertoire `model/colis/ocaml/tests` du dépôt Git.

Lors de cette campagne de tests, on a identifié trois incohérences. La première incohérence concerne la boucle **do while**. La deuxième concerne le shell builtin **shift**. Enfin la troisième concerne la substitution de commandes (spécifiée par le mot-clé "embed" dans la syntaxe concrète de CoLiS).

Les tests qui mettent en évidence ces erreurs sont présentés sur la figure 11.

8.3 Erreur n°1 : Shift

Le builtin **shift** permet de déplacer le "curseur" de positionement des arguments vers la droite (d'un pas vers la droite par défaut, de n pas si n est explicité en argument de **shift**). Lorsque la variable `$0` contient le dernier argument, le shell `dash` interprète le **shift** en générant un message d'erreur (`shift: can't shift that many`), avec un comportement **Fatal** (code de sortie= 2). Quant au shell `bash`, il

12. La sortie d'erreur est ignorée.

programme CoLiS	script shell traduit depuis CoLiS # !/bin/sh set -e	résultat de l'exécution du : [13]	
		programme CoLiS via l'interpréteur CoLiS	script correspondant via le shell [14]
		true	false
function args = do call [{'echo'} ; {arg 0}] while shift done begin call [{'echo'} ; {arg 0}] ; call [{'args'} ; {'un'} ; {'deux'} ; {'trois'}] ; call [split {'echo fin'}] end	args () { while : ; do echo "${1}" ; shift \|\| break ; done } echo "${1}" ; args un deux trois ; echo fin	1 2 un 3 deux 4 trois 5 6 fin	1 2 un 3 deux 4 trois 5
		false	true
do false while fatal done	while : ; do ! true ; false || break ; done	*(vide)*	*(vide)*
		true	true
var x : string begin x := {embed call [{'date'} ; {'+%H'}]} ; call [{'echo'} ; {'heure : '} ; {x}] end	x=$(date +%H) ; echo 'heure : "'$x"	1 heure : 12 2	1 heure : 12

FIGURE 11 – Tests révèlant des erreurs

l'interprète avec un comportement **Fatal** sans générer de message sur la sortie d'erreur (code de sortie= 1). Cependant, en mode strict, il semble que le script `shift || true` interrompt le script si il est interprété par le shell `dash` mais pas par le shell `bash`. Et pourtant, toujours en mode strict, le script `false || true` n'interrompt le script ni avec `dash`, ni avec `bash`. Il semble que l'interpréteur CoLiS reproduit le comportement de `bash` plutôt que de `dash` (les deux sont compatibles avec la Debian Policy [15]) alors que le traducteur CoLiS/shell générait les scripts avec l'en-tête `#!/bin/sh`. Cela est désormais corrigé. Le choix de `bash` semble effectivement plus judicieux au vue de la stabilité du mode strict, du moins dans la cohérence de l'interprétation de `shift || true` et `false || true` quand shift est **Fatal** (le `false` du shell étant **Fatal**).

8.4 Erreur n°2 : Boucle do while

Le comportement du terme **do false while fatal** est **false** pour l'interpréteur CoLiS, **true** pour le shell. Le bug se situe au niveau de la traduction CoLiS/shell. En effet, le terme **do** t_1 **while** t_2 de CoLiS est traduit en shell par `while : ; do` <t_1 `; ` <t_2> `|| break; done`, la boucle **do while** n'existant pas telle quelle en shell. Or, le **break** fait tout le temps sortir la boucle avec un comportement **true**, ce qui n'est

15. https://www.debian.org/doc/debian-policy/ch-files.html#s-scripts

pas le cas d'une boucle **do while**, dont le code de sortie est celui de la dernière exécution du corps de la boucle (le **break** n'étant qu'un ajout artificiel). Une retranscription fidèle du terme **do** t_1 **while** t_2 est `<`t_1`> while <`t_2`>; do <`t_1`> ; done`. Son inconvénient est le fait de devoir dupliquer le code du corps de la boucle, ce qui n'est pas une solution élégante. Cette solution a désormais été adoptée pour la traduction du **do while**, mais encore mieux, le terme **while** a été ajouté en CoLiS.

8.5 Erreur n°3 : Substitution de commandes

La substitution de commandes permet d'exécuter dans un sous-shell – donc dans un autre environnement – une commande et manipuler sa sortie standard comme une valeur. On réalise cela via l'instruction suivante (si on souhaite par exemple affecter la valeur résultante à une variable *var*) : `<`*var*`>=$(<`*commande*`>)`. L'échec du test vient du fait que l'interpréteur CoLiS gère les sauts de ligne de la sortie standard de la commande substituée de la même manière que ceux de la sortie standard de l'environnement courant. Illustration avec `printf` qui n'ajoute pas de saut de ligne par défaut, et `echo` qui ajoute un saut de ligne par défaut :

```
> printf bonjour
bonjour> echo bonjour
bonjour
> x=$(printf bonjour)
> y=$(echo bonjour)
> z=$(printf "bonjour\n\n\n")
> a=$(printf "\nbonjour\n\n\n")
> if [ "$x" = "$y" ] ; then echo ok ; else echo diff ; fi
ok
> if [ "$x" = "$z" ] ; then echo ok ; else echo diff ; fi
ok
> if [ "$x" = "$a" ] ; then echo ok ; else echo diff ; fi
diff
```

Il semble que le shell (`bash`) retire systématiquement les sauts de ligne situés à la fin de la sortie standard récoltée dans la substitution de commande, ce que l'interpréteur CoLiS ne prenait pas en compte, bien qu'il fût prouvé. En effet, l'erreur venait de l'implémentation des règles sémantiques du langage CoLiS. Or l'interpréteur fût prouvé par rapport à cette implémentation. La sémantique du langage CoLiS devra donc être corrigée en conséquence.

9 Conclusions et travaux futurs

Un traducteur de programmes CoLiS vers des scripts shell a été conçu ainsi que des outils permettant de réaliser des tests. Le tout a notamment permis de révéler une erreur dans la définition de la sémantique de CoLiS.

Une règle supplémentaire a été ajouté à la sémantique du langage CoLiS, ainsi que le terme correspondant **while** à sa syntaxe abstraite, afin de retranscrire plus directement le `while` du shell, celui-ci étant auparavant représenté par une combinaison des termes **if** et **do while**.

9.1 Travaux futurs : correction de bugs et extension des tests

Parmi les bugs identifiés, il reste donc à corriger la sémantique de l'appel imbriqué de commandes, qui devrait donc supprimer les sauts de lignes de fin, et de modifier également l'interpréteur CoLiS en ce sens.

La couverture de l'ensemble des traits du langage CoLiS n'est pas assurée. Par exemple, le traitement des pipes, et les commandes utilisant leur entrée standard, n'a été que très peu testée. L'élargissement du jeu de tests est donc souhaitable, ainsi qu'une mesure précise de leur couverture du langage.

9.2 Travaux futurs : outillage supplémentaire

La réalisation d'un traducteur dans le sens inverse, c'est-à-dire du shell vers CoLiS, reste encore à être développée au sein du projet. L'une des principales difficultés est de procéder à une inférence de type : chaque variable devra être déclarée au préalable, avec un type adéquat (string ou list). En cas d'utilisation d'une même variable à la fois en tant que liste et que string, une erreur devra être signalée, car cela peut signifier une erreur importante dans le script shell de départ.

Il est aussi envisagé d'implémenter le `break` du shell dans le langage CoLiS car son comportement est particulier[7].

La comparaison des résultats de l'interpréteur CoLiS et du shell n'est effectué que par rapport à la sortie standard et au behavior de retour. Il faudra également comparer les changements effectués dans le système de fichiers. Un préalable est de remplacer l'interprétation des commandes builtins par l'interpréteur CoLiS actuel via le shell, par des actions sur un modèle du système de fichiers, à écrire en Why3 ou en OCaml.

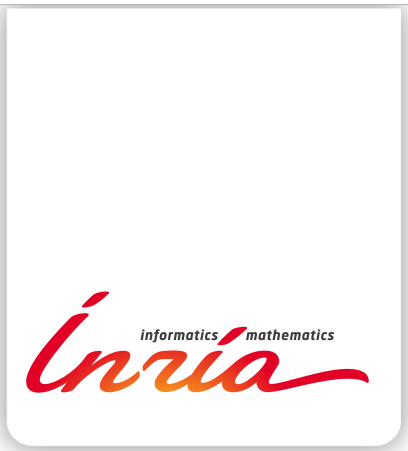
Références

[1] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6) :709–727, 2015. See also http://toccata.lri.fr/gallery/fm2012comp.en.html.

[2] N. Jeannerod. Le coquillage dans le CoLiS-mateur : formalisation d'un langage de programmation de type Shell. In *Vingt-huitièmes Journées Francophones des Langages Applicatifs*, Gourette, France, Jan. 2017.

[3] N. Jeannerod, C. Marché, and R. Treinen. A formally verified interpreter for a shell-like programming language. In A. Paskevich and T. Wies, editors, *9th Working Conference on Verified Software : Theories, Tools and Experiments (VSTTE)*, Lecture Notes in Computer Science, Heidelberg, Germany, July 2017. Springer.

[4] N. Jeannerod, Y. Régis-Gianas, and R. Treinen. Having fun with 31.521 Shell scripts. https://hal.archives-ouvertes.fr/hal-01513750, 2017.



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803