A Formally Verified Interpreter for a Shell-like Programming Language

Claude Marché Nicolas Jeannerod Ralf Treinen

INSTITUT
DE RECHERCHE
EN INFORMATIQUE
FONDAMENTALE

Vals seminar, July 7, 2017

The CoLiS project

- Correctness of Linux Scripts
- ANR project, 5 years (October 2015 – September 2020)
- Three workpackages:
  - IRIF, Université Paris-Diderot
  - Inria Saclay
  - Inria Lille

- **Goal:** apply verification techniques to *Debian maintainer scripts*. Those are POSIX Shell scripts:
  - used for installation, upgrade, removal of packages
  - ran as *root* user
  - mistakes are easy to make and hard to detect

- We are not trying to replace the Shell.

The CoLiS project

- Correctness of Linux Scripts
- ANR project, 5 years (October 2015 – September 2020)
- Three workpackages:
    - IRIF, Université Paris-Diderot
    - Inria Saclay
    - Inria Lille
- **Goal:** apply verification techniques to *Debian maintainer scripts*. Those are POSIX Shell scripts:
    - used for installation, upgrade, removal of packages
    - ran as *root* user
    - mistakes are easy to make and hard to detect
- We are not trying to replace the Shell.

The CoLiS project

- Correctness of Linux Scripts
- ANR project, 5 years (October 2015 – September 2020)
- Three workpackages:
    - IRIF, Université Paris-Diderot
    - Inria Saclay
    - Inria Lille

- **Goal:** apply verification techniques to *Debian maintainer scripts*. Those are POSIX Shell scripts:
    - used for installation, upgrade, removal of packages
    - ran as *root* user
    - mistakes are easy to make and hard to detect

- We are not trying to replace the Shell.

The CoLiS project

- Correctness of Linux Scripts
- ANR project, 5 years (October 2015 – September 2020)
- Three workpackages:
  - IRIF, Université Paris-Diderot
  - Inria Saclay
  - Inria Lille
- **Goal:** apply verification techniques to *Debian maintainer scripts*. Those are POSIX Shell scripts:
  - used for installation, upgrade, removal of packages
  - ran as *root* user
  - mistakes are easy to make and hard to detect
- We are not trying to replace the Shell.

The CoLiS project

- Correctness of Linux Scripts
- ANR project, 5 years (October 2015 – September 2020)
- Three workpackages:
  - IRIF, Université Paris-Diderot
  - Inria Saclay
  - Inria Lille
- **Goal:** apply verification techniques to *Debian maintainer scripts*. Those are POSIX Shell scripts:
  - used for installation, upgrade, removal of packages
  - ran as *root* user
  - mistakes are easy to make and hard to detect
- We are not trying to replace the Shell.

The CoLiS project

- Correctness of Linux Scripts
- ANR project, 5 years (October 2015 – September 2020)
- Three workpackages:
    - IRIF, Université Paris-Diderot
    - Inria Saclay
    - Inria Lille
- **Goal:** apply verification techniques to *Debian maintainer scripts*. Those are POSIX Shell scripts:
    - used for installation, upgrade, removal of packages
    - ran as *root* user
    - mistakes are easy to make and hard to detect
- We are not trying to replace the Shell.

The CoLiS project

- Correctness of Linux Scripts
- ANR project, 5 years (October 2015 – September 2020)
- Three workpackages:
    - IRIF, Université Paris-Diderot
    - Inria Saclay
    - Inria Lille

- **Goal:** apply verification techniques to *Debian maintainer scripts*. Those are POSIX Shell scripts:
    - used for installation, upgrade, removal of packages
    - ran as *root* user
    - mistakes are easy to make and hard to detect

- We are not trying to replace the Shell.

The CoLiS project

- Correctness of Linux Scripts
- ANR project, 5 years (October 2015 – September 2020)
- Three workpackages:
  - IRIF, Université Paris-Diderot
  - Inria Saclay
  - Inria Lille

- **Goal:** apply verification techniques to *Debian maintainer scripts*. Those are POSIX Shell scripts:
  - used for installation, upgrade, removal of packages
  - ran as *root* user
  - mistakes are easy to make and hard to detect

- We are not trying to replace the Shell.

Table of Contents

Execute arbitrary strings

Execute commands from strings:

```
a="echo foo"
$a                  ## echoes "foo"
```

or any code with eval:

```
eval "if true; then echo foo; fi"
```

Execute arbitrary strings

Execute commands from strings:

```
a="echo foo"
$a                    ## echoes "foo"
```

or any code with eval:

```
eval "if true; then echo foo; fi"
```

Dynamic

Everything is dynamic:

```
f () { g; }
g () { a=bar; }
a=foo
f
echo $a              ## echoes "bar"
```

Example 2-in-1 (expansion and dynamic scoping):

```
f () { echo $1 $a; }
a=foo
a=bar f $a           ## echoes "foo bar"
echo $a              ## echoes "bar"
```

Dynamic

Everything is dynamic:

```
f () { g; }
g () { a=bar; }
a=foo
f
echo $a            ## echoes "bar"
```

Example 2-in-1 (expansion and dynamic scoping):

```
f () { echo $1 $a; }
a=foo
a=bar f $a         ## echoes "foo bar"
echo $a            ## echoes "bar"
```

Dynamic

Everything is dynamic:

```
f () { g; }
g () { a=bar; }
a=foo
f
echo $a              ## echoes "bar"
```

Example 2-in-1 (expansion and dynamic scoping):

```
f () { echo $1 $a; }
a=foo
a=bar f $a           ## echoes "foo bar"
echo $a              ## echoes "bar"
```

Behaviours

Nice falses and the violent one:

```
set -e
! true ; echo foo  ## echoes "foo"
false  ; echo foo  ## exits
```

Many ways to catch “exit” and “return”:

```
( exit )
( return )
exit | true
echo "still not dead"
exit
```

Behaviours

Nice falses and the violent one:

```
set -e
! true ; echo foo  ## echoes "foo"
false  ; echo foo  ## exits
```

Many ways to catch “exit” and “return”:

```
( exit )
( return )
exit | true
echo "still not dead"
exit
```

Behaviours

Nice falses and the violent one:

```
set -e
! true ; echo foo  ## echoes "foo"
false  ; echo foo  ## exits
```

Many ways to catch "exit" and "return":

```
( exit )
( return )
exit | true
echo "still not dead"
exit
```

How behaviours are handled

	0	1-127	1-127*	Return 0	Return 1-127	Exit 0	Exit 1-127
Pipe	Normal						
Sequence	Normal		Exception				
Test	True	False		Exception			
Function call	Success	Failure		Success	Failure	Exception	
Subprocess	Success	Failure		Success	Failure	Success	Failure

The expansion mechanism

Used to represent both strings and lists of strings:

```
args="-l -a"
args="$args -h"
path=/home
path=$path/nicolas
ls $args $path
```

Can contain all sorts of things:

```
echo foo$(echo "$bar"baz)"$bar"
```

The expansion mechanism

Used to represent both strings and lists of strings:

```
args="-l -a"
args="$args -h"
path=/home
path=$path/nicolas
ls $args $path
```

Can contain all sorts of things:

```
echo foo$(echo "$bar"baz)"$bar"
```

The expansion mechanism

Used to represent both strings and lists of strings:

```
args="-l -a"
args="$args -h"
path=/home
path=$path/nicolas
ls $args $path
```

Can contain all sorts of things:

```
echo foo$(echo "$bar"baz)"$bar"
```

The expansion mechanism

Used to represent both strings and lists of strings:

```
args="-l -a"
args="$args -h"
path=/home
path=$path/nicolas
ls $args $path
```

Can contain all sorts of things:

```
echo foo$(echo "$bar"baz)"$bar"
```

Table of Contents

Requirements

- Intermediary language (not a replacement of Shell);
- "Cleaner" than Shell (no `eval` for instance);
- Well-defined and easily understandable semantics:
  - Some typing (strings vs. string lists),
  - Variables and functions declared in a header,
  - Dangers made more explicit;
- "Close enough" to Shell:
  - A reader must be convinced that it shares the same semantics as the Shell,
  - Target of an automated translation from Shell.

Requirements

- Intermediary language (not a replacement of Shell);
- "Cleaner" than Shell (no `eval` for instance);
- Well-defined and easily understandable semantics:
  - Some typing (strings vs. string lists),
  - Variables and functions declared in a header,
  - Dangers made more explicit;
- "Close enough" to Shell:
  - A reader must be convinced that it shares the same semantics as the Shell,
  - Target of an automated translation from Shell.

Requirements

- Intermediary language (not a replacement of Shell);
- "Cleaner" than Shell (no `eval` for instance);
- Well-defined and easily understandable semantics:
  - Some typing (strings vs. string lists),
  - Variables and functions declared in a header,
  - Dangers made more explicit;
- "Close enough" to Shell:
  - A reader must be convinced that it shares the same semantics as the Shell,
  - Target of an automated translation from Shell.

Requirements

- Intermediary language (not a replacement of Shell);
- “Cleaner” than Shell (no `eval` for instance);
- Well-defined and easily understandable semantics:
  - Some typing (strings vs. string lists),
  - Variables and functions declared in a header,
  - Dangers made more explicit;
- “Close enough” to Shell:
  - A reader must be convinced that it shares the same semantics as the Shell,
  - Target of an automated translation from Shell.

Syntax – 1

String variables	x_s	$\in$	$SVar$
List variables	x_l	$\in$	$LVar$
Procedures names	c	$\in$	$\mathcal{F}$

Programs	p	$::=$	$vdecl^*$ $pdecl^*$ **program** t
Variables decl.	$vdecl$	$::=$	**varstring** x_s $\mid$ **varlist** x_l
Procedures decl.	$pdecl$	$::=$	**proc** c **is** t

Syntax – 2

$$
\begin{array}{llcl}
\text{Terms} & t & ::= & \textbf{true} \quad | \quad \textbf{false} \quad | \quad \textbf{fatal} \\
& & | & \textbf{return}\ t \quad | \quad \textbf{exit}\ t \\
& & | & x_s := s \quad | \quad x_l := l \\
& & | & t\ ;\ t \quad | \quad \textbf{if}\ t\ \textbf{then}\ t\ \textbf{else}\ t \\
& & | & \textbf{for}\ x_s\ \textbf{in}\ l\ \textbf{do}\ t \quad | \quad \textbf{while}\ t\ \textbf{do}\ t \\
& & | & \textbf{process}\ t \quad | \quad \textbf{pipe}\ t\ \textbf{into}\ t \\
& & | & \textbf{call}\ l \quad | \quad \textbf{shift}
\end{array}
$$

Syntax – 3

$$
\begin{array}{llcl}
\text{String expressions} & s & ::= & \mathbf{nil}_s \mid f_s :: s \\
\text{String fragments} & f_s & ::= & \sigma \mid x_s \mid n \mid t \\
\\
\text{List expressions} & l & ::= & \mathbf{nil}_l \mid f_l :: l \\
\text{List fragments} & f_l & ::= & s \mid \mathbf{split}\ s \mid x_l
\end{array}
$$

Semantics – First definitions

$$
\begin{array}{lcll}
\text{Behaviours: terms} & b & \in & \{\mathsf{True}, \mathsf{False}, \mathsf{Fatal}, \mathsf{Return\ True} \\
 & & & \ \mathsf{Return\ False}, \mathsf{Exit\ True}, \mathsf{Exit\ False}\} \\
\text{Behaviours: expressions} & \beta & \in & \{\mathsf{True}, \mathsf{Fatal}, \mathsf{None}\} \\
\text{Environments: strings} & & & \mathit{SEnv} \triangleq [\mathit{SVar} \rightharpoonup \mathit{String}] \\
\text{Environments: lists} & & & \mathit{LEnv} \triangleq [\mathit{LVar} \rightharpoonup \mathit{StringList}] \\
\text{Contexts} & \Gamma & \in & \mathcal{FS} \times \mathit{String} \times \mathit{StringList} \\
 & & & \quad \times \mathit{SEnv} \times \mathit{LEnv}
\end{array}
$$

In a context: file system, standard input, arguments line, string environment, list environment.

Semantics – First definitions

$$
\begin{array}{lcll}
\text{Behaviours: terms} & \mathsf{b} & \in & \{\mathsf{True}, \mathsf{False}, \mathsf{Fatal}, \mathsf{Return\ True} \\
 & & & \ \mathsf{Return\ False}, \mathsf{Exit\ True}, \mathsf{Exit\ False}\} \\
\text{Behaviours: expressions} & \beta & \in & \{\mathsf{True}, \mathsf{Fatal}, \mathsf{None}\} \\
\\
\text{Environments: strings} & & & \mathit{SEnv} \triangleq [\mathit{SVar} \rightharpoonup \mathit{String}] \\
\text{Environments: lists} & & & \mathit{LEnv} \triangleq [\mathit{LVar} \rightharpoonup \mathit{StringList}] \\
\text{Contexts} & \Gamma & \in & \mathcal{FS} \times \mathit{String} \times \mathit{StringList} \\
 & & & \quad \times \mathit{SEnv} \times \mathit{LEnv}
\end{array}
$$

In a context: file system, standard input, arguments line, string environment, list environment.

Semantic judgments

Judgments: terms $t_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma'}$

Judgments: string fragment $f_{s/\Gamma} \Downarrow_{sf} \sigma \star \beta_{/\Gamma'}$

Judgments: string expression $s_{/\Gamma} \Downarrow_{s} \sigma \star \beta_{/\Gamma'}$

Judgments: list fragment $f_{l/\Gamma} \Downarrow_{lf} \lambda \star \beta_{/\Gamma'}$

Judgments: list expression $l_{/\Gamma} \Downarrow_{l} \lambda \star \beta_{/\Gamma'}$

A few rules – Sequence

$$\textsc{Sequence-Normal}\quad \frac{{t_1}_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1} \qquad b_1 \in \{\mathsf{True}, \mathsf{False}\} \qquad {t_2}_{/\Gamma_1} \Downarrow \sigma_2 \star {b_2}_{/\Gamma_2}}{(t_1\ ;\ t_2)_{/\Gamma} \Downarrow \sigma_1\sigma_2 \star {b_2}_{/\Gamma_2}}$$

$$\textsc{Sequence-Exception}\quad \frac{{t_1}_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1} \qquad b_1 \in \{\mathsf{Fatal}, \mathsf{Return}\ \_, \mathsf{Exit}\ \_\}}{(t_1\ ;\ t_2)_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1}}$$

A few rules – Sequence

$$\textsc{Sequence-Normal}\ \dfrac{{t_1}_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1} \qquad b_1 \in \{\mathsf{True}, \mathsf{False}\} \qquad {t_2}_{/\Gamma_1} \Downarrow \sigma_2 \star {b_2}_{/\Gamma_2}}{(t_1\ ;\ t_2)_{/\Gamma} \Downarrow \sigma_1\sigma_2 \star {b_2}_{/\Gamma_2}}$$

$$\textsc{Sequence-Exception}\ \dfrac{{t_1}_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1} \qquad b_1 \in \{\mathsf{Fatal}, \mathsf{Return}\ \_, \mathsf{Exit}\ \_\}}{(t_1\ ;\ t_2)_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1}}$$

A few rules – Branching

$$\dfrac{\textsc{Branching-True} \\ t_1{}_{/\Gamma} \Downarrow \sigma_1 \star b_1{}_{/\Gamma_1} \qquad b_1 = \mathsf{True} \qquad t_2{}_{/\Gamma_2} \Downarrow \sigma_2 \star b_2{}_{/\Gamma_2}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1\sigma_2 \star b_2{}_{/\Gamma_2}}$$

$$\dfrac{\textsc{Branching-False} \\ t_1{}_{/\Gamma} \Downarrow \sigma_1 \star b_1{}_{/\Gamma_1} \qquad b_1 \in \{\mathsf{False}, \mathsf{Fatal}\} \qquad t_3{}_{/\Gamma_3} \Downarrow \sigma_3 \star b_3{}_{/\Gamma_3}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1\sigma_3 \star b_3{}_{/\Gamma_3}}$$

$$\dfrac{\textsc{Branching-Exception} \\ t_1{}_{/\Gamma} \Downarrow \sigma_1 \star b_1{}_{/\Gamma_1} \qquad b_1 \in \{\mathsf{Return}\ \_, \mathsf{Exit}\ \_\}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1 \star b_1{}_{/\Gamma_1}}$$

A few rules – Branching

$$\textsc{Branching-True}\quad \frac{{t_1}_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1} \qquad b_1 = \mathsf{True} \qquad {t_2}_{/\Gamma_2} \Downarrow \sigma_2 \star {b_2}_{/\Gamma_2}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1\sigma_2 \star {b_2}_{/\Gamma_2}}$$

$$\textsc{Branching-False}\quad \frac{{t_1}_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1} \qquad b_1 \in \{\mathsf{False}, \mathsf{Fatal}\} \qquad {t_3}_{/\Gamma_3} \Downarrow \sigma_3 \star {b_3}_{/\Gamma_3}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1\sigma_3 \star {b_3}_{/\Gamma_3}}$$

$$\textsc{Branching-Exception}\quad \frac{{t_1}_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1} \qquad b_1 \in \{\mathsf{Return}\ \_, \mathsf{Exit}\ \_\}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1}}$$

A few rules – Branching

$$\textsc{Branching-True}\quad \frac{{t_1}_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1} \qquad b_1 = \mathsf{True} \qquad {t_2}_{/\Gamma_2} \Downarrow \sigma_2 \star {b_2}_{/\Gamma_2}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1\sigma_2 \star {b_2}_{/\Gamma_2}}$$

$$\textsc{Branching-False}\quad \frac{{t_1}_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1} \qquad b_1 \in \{\mathsf{False}, \mathsf{Fatal}\} \qquad {t_3}_{/\Gamma_3} \Downarrow \sigma_3 \star {b_3}_{/\Gamma_3}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1\sigma_3 \star {b_3}_{/\Gamma_3}}$$

$$\textsc{Branching-Exception}\quad \frac{{t_1}_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1} \qquad b_1 \in \{\mathsf{Return}\ \_, \mathsf{Exit}\ \_\}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1}}$$

How behaviours are handled

	True	False	Fatal	Return True	Return False	Exit True	Exit False
Pipe	Normal						
Sequence	Normal		Exception				
Test	True	False		Exception			
Function call	Success	Failure		Success	Failure	Exception	
Subprocess	Success	Failure		Success	Failure	Success	Failure

A few rules – Mutual recursion

Terms depend on string expressions:

$$\textsc{Assignment-String}\quad \frac{s_{/\Gamma} \Downarrow_s \sigma \star \beta_{/\Gamma'}}{x := s_{/\Gamma} \Downarrow \text{``''} \star \beta_{/\Gamma'[\mathtt{senv}=\Gamma'.\mathtt{senv}[x\leftarrow\sigma]]}}$$

and string fragments depend on terms:

$$\textsc{String-Subprocess}\quad \frac{t_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma'}}{t_{/\Gamma} \Downarrow_{sf} \sigma \star \overline{b}_{/\Gamma'}}$$

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

A few rules – Mutual recursion

Terms depend on string expressions:

$$\textsc{Assignment-String}\quad \frac{s_{/\Gamma} \Downarrow_s \sigma \star \beta_{/\Gamma'}}{x := s_{/\Gamma} \Downarrow \text{``''} \star \beta_{/\Gamma'[\mathtt{senv}=\Gamma'.\mathtt{senv}[x\leftarrow\sigma]]}}$$

and string fragments depend on terms:

$$\textsc{String-Subprocess}\quad \frac{t_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma'}}{t_{/\Gamma} \Downarrow_{sf} \sigma \star \overline{b}_{/\Gamma'}}$$

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

Table of Contents

Why3

- Platform for deductive program verification;
- WhyML: language for both specification and programming;
- Standard library:
  - integer arithmetic,
  - boolean operations,
  - maps,
  - etc.;
- Support of imperative traits:
  - references,
  - exceptions,
  - while and for loops;
- Proof obligations are given to external theorem provers;
- Possibility to extract WhyML code to OCaml.

Why3

- Platform for deductive program verification;
- WhyML: language for both specification and programming;
- Standard library:
  - integer arithmetic,
  - boolean operations,
  - maps,
  - etc.;
- Support of imperative traits:
  - references,
  - exceptions,
  - while and for loops;
- Proof obligations are given to external theorem provers;
- Possibility to extract WhyML code to OCaml.

Why3

- Platform for deductive program verification;
- WhyML: language for both specification and programming;
- Standard library:
  - integer arithmetic,
  - boolean operations,
  - maps,
  - etc.;
- Support of imperative traits:
  - references,
  - exceptions,
  - while and for loops;
- Proof obligations are given to external theorem provers;
- Possibility to extract WhyML code to OCaml.

Why3

- Platform for deductive program verification;

- WhyML: language for both specification and programming;
- Standard library:
  - integer arithmetic,
  - boolean operations,
  - maps,
  - etc.;
- Support of imperative traits:
  - references,
  - exceptions,
  - while and for loops;

- Proof obligations are given to external theorem provers;

- Possibility to extract WhyML code to OCaml.

Why3

- Platform for deductive program verification;

- WhyML: language for both specification and programming;
- Standard library:
    - integer arithmetic,
    - boolean operations,
    - maps,
    - etc.;
- Support of imperative traits:
    - references,
    - exceptions,
    - while and for loops;

- Proof obligations are given to external theorem provers;
- Possibility to extract WhyML code to OCaml.

Why3

- Platform for deductive program verification;

- WhyML: language for both specification and programming;
- Standard library:
    - integer arithmetic,
    - boolean operations,
    - maps,
    - etc.;
- Support of imperative traits:
    - references,
    - exceptions,
    - while and for loops;

- Proof obligations are given to external theorem provers;
- Possibility to extract WhyML code to OCaml.

Syntax

```
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
  | 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
```

Semantic judgments

```
inductive eval_term term context
                    string behaviour context

with eval_sexpr sexpr context
                string bool context

with eval_sfrag sfrag context
                string (option bool) context

with eval_lexpr lexpr context
                (list string) bool context

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

A few rules – Sequence

```
| EvalT_Seq_Normal : forall t1 Γ σ1 b1 Γ1 t2 σ2 b2 Γ2.
  eval_term t1 Γ σ1 (BNormal b1) Γ1 ->
  eval_term t2 Γ1 σ2 b2 Γ2 ->
  eval_term (TSeq t1 t2) Γ (concat σ1 σ2) b2 Γ2
```

```
| EvalT_Seq_Error : forall t1 Γ σ1 b1 Γ1 t2.
  eval_term t1 Γ σ1 b1 Γ1 ->
  (match b1 with BNormal _ -> false | _ -> true end) ->
  eval_term (TSeq t1 t2) Γ σ1 b1 Γ1
```

A few rules – Sequence

```
| EvalT_Seq_Normal : forall t1 Γ σ1 b1 Γ1 t2 σ2 b2 Γ2.
  eval_term t1 Γ σ1 (BNormal b1) Γ1 ->
  eval_term t2 Γ1 σ2 b2 Γ2 ->
  eval_term (TSeq t1 t2) Γ (concat σ1 σ2) b2 Γ2
```

```
| EvalT_Seq_Error : forall t1 Γ σ1 b1 Γ1 t2.
  eval_term t1 Γ σ1 b1 Γ1 ->
  (match b1 with BNormal _ -> false | _ -> true end) ->
  eval_term (TSeq t1 t2) Γ σ1 b1 Γ1
```

A few rules – Branching

```
| EvalT_If_True : forall t1 Γ σ1 Γ1 t2 σ2 b2 Γ2 t3.
  eval_term t1 Γ σ1 (BNormal True) Γ1 ->
  eval_term t2 Γ1 σ2 b2 Γ2 ->
  eval_term (TIf t1 t2 t3) Γ (concat σ1 σ2) b2 Γ2

| EvalT_If_False : forall t1 Γ σ1 b1 Γ1 t3 σ3 b3 Γ3 t2.
  eval_term t1 Γ σ1 b1 Γ1 ->
  (match b1 with BNormal False | BFatal -> true | _ -> false en
  eval_term t3 Γ1 σ3 b3 Γ3 ->
  eval_term (TIf t1 t2 t3) Γ (concat σ1 σ3) b3 Γ3

| EvalT_If_Transmit : forall t1 Γ σ1 b1 Γ1 t2 t3.
  eval_term t1 Γ σ1 b1 Γ1 ->
  (match b1 with BReturn _ | BExit _ -> true | _ -> false end)
  eval_term (TIf t1 t2 t3) Γ σ1 b1 Γ1
```

A few rules – Branching

```
| EvalT_If_True : forall t1 Γ σ1 Γ1 t2 σ2 b2 Γ2 t3.
  eval_term t1 Γ σ1 (BNormal True) Γ1 ->
  eval_term t2 Γ1 σ2 b2 Γ2 ->
  eval_term (TIf t1 t2 t3) Γ (concat σ1 σ2) b2 Γ2
```

```
| EvalT_If_False : forall t1 Γ σ1 b1 Γ1 t3 σ3 b3 Γ3 t2.
  eval_term t1 Γ σ1 b1 Γ1 ->
  (match b1 with BNormal False | BFatal -> true | _ -> false en
  eval_term t3 Γ1 σ3 b3 Γ3 ->
  eval_term (TIf t1 t2 t3) Γ (concat σ1 σ3) b3 Γ3
```

```
| EvalT_If_Transmit : forall t1 Γ σ1 b1 Γ1 t2 t3.
  eval_term t1 Γ σ1 b1 Γ1 ->
  (match b1 with BReturn _ | BExit _ -> true | _ -> false end)
  eval_term (TIf t1 t2 t3) Γ σ1 b1 Γ1
```

A few rules – Branching

```
| EvalT_If_True : forall t1 Γ σ1 Γ1 t2 σ2 b2 Γ2 t3.
  eval_term t1 Γ σ1 (BNormal True) Γ1 ->
  eval_term t2 Γ1 σ2 b2 Γ2 ->
  eval_term (TIf t1 t2 t3) Γ (concat σ1 σ2) b2 Γ2


| EvalT_If_False : forall t1 Γ σ1 b1 Γ1 t3 σ3 b3 Γ3 t2.
  eval_term t1 Γ σ1 b1 Γ1 ->
  (match b1 with BNormal False | BFatal -> true | _ -> false en
  eval_term t3 Γ1 σ3 b3 Γ3 ->
  eval_term (TIf t1 t2 t3) Γ (concat σ1 σ3) b3 Γ3


| EvalT_If_Transmit : forall t1 Γ σ1 b1 Γ1 t2 t3.
  eval_term t1 Γ σ1 b1 Γ1 ->
  (match b1 with BReturn _ | BExit _ -> true | _ -> false end) -
  eval_term (TIf t1 t2 t3) Γ σ1 b1 Γ1
```

A few rules – Mutual recursion

```
| EvalT_AsString : forall s Γ σ β Γ' Γ'' x_s.
  eval_sexpr s Γ σ β Γ' ->
  Γ'' = update_senv Γ' x_s σ ->
  eval_term (TAsString x_s s) Γ empty_string
    (if β then BNormal True else BFatal) Γ''
```

```
| EvalSF_Process : forall 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 Tr
    {Γ with c_fs = Γ'.c_fs ; c_input = Γ'.c_input}
```

A few rules – Mutual recursion

```
| EvalT_AsString : forall s Γ σ β Γ' Γ'' x_s.
  eval_sexpr s Γ σ β Γ' ->
  Γ'' = update_senv Γ' x_s σ ->
  eval_term (TAsString x_s s) Γ empty_string
    (if β then BNormal True else BFatal) Γ''
```

```
| EvalSF_Process : forall 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 Tru
    {Γ with c_fs = Γ'.c_fs ; c_input = Γ'.c_input}
```

Table of Contents

Why?

- For fun;
- Helps detecting the potential mistakes;
- We can compare the observational behaviour of our interpreter with known implementations of the POSIX Shell;
- It gives us a way to test an automated translation from Shell to CoLiS.

Why?

- For fun;
- Helps detecting the potential mistakes;
- We can compare the observational behaviour of our interpreter with known implementations of the POSIX Shell;
- It gives us a way to test an automated translation from Shell to CoLiS.

Why?

- For fun;
- Helps detecting the potential mistakes;
- We can compare the observational behaviour of our interpreter with known implementations of the POSIX Shell;
- It gives us a way to test an automated translation from Shell to CoLiS.

Why?

- For fun;
- Helps detecting the potential mistakes;
- We can compare the observational behaviour of our interpreter with known implementations of the POSIX Shell;
- It gives us a way to test an automated translation from Shell to CoLiS.

Table of Contents

Spirit of the code

- Set of mutually recursive functions;
- ML-style with imperative traits;
- Fatal, Return _ and Exit _ are exceptions;
- `stdout` is a reference.

```
exception EFatal context
exception EReturn (bool,context)
exception EExit (bool,context)

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

with interp_sexpr_aux (s: sexpr) (Γ: context) (previous: bool)
                      : (string, bool, context)

with interp_sfrag_aux (fs: sfrag) (Γ: context) (previous: bool)
                      : (string, bool, context)

...
```

Spirit of the code

- Set of mutually recursive functions;
- ML-style with imperative traits;
- Fatal, Return _ and Exit _ are exceptions;
- `stdout` is a reference.

```
exception EFatal context
exception EReturn (bool,context)
exception EExit (bool,context)

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

with interp_sexpr_aux (s: sexpr) (Γ: context) (previous: bool)
                      : (string, bool, context)

with interp_sfrag_aux (fs: sfrag) (Γ: context) (previous: bool)
                      : (string, bool, context)
...
```

Body – Sequence and branching

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

  | TSeq t₁ t₂ ->
    let (_, Γ₁) = interp_term t₁ Γ stdout in
    interp_term t₂ Γ₁ stdout

  | TIf t₁ t₂ t₃ ->
    let (b₁, Γ₁) =
      try
        interp_term t₁ Γ stdout
      with
        EFatal Γ₁ -> (false, Γ₁)
      end
    in
    interp_term (if b₁ then t₂ else t₃) Γ₁ stdout
```

Body – Sequence and branching

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

  | TSeq t₁ t₂ ->
    let (_, Γ₁) = interp_term t₁ Γ stdout in
    interp_term t₂ Γ₁ stdout

  | TIf t₁ t₂ t₃ ->
    let (b₁, Γ₁) =
      try
        interp_term t₁ Γ stdout
      with
        EFatal Γ₁ -> (false, Γ₁)
      end
    in
    interp_term (if b₁ then t₂ else t₃) Γ₁ stdout
```

Body – Mutual recursion

```
let rec interp_term (t: term) (Γ: context)
                    (stdout: ref string) : (bool, context)
=
  match t with
  | TAsString xs s ->
    let (σ, b, Γ') = interp_sexpr s Γ in
    let Γ'' = update_senv Γ' xs σ in
    if b then (true, Γ'') else raise (EFatal Γ'')
  ...
```

```
with interp_sfrag_aux (f_s: sfrag) (Γ: context) (previous: bool)
                      : (string, bool, context)
=
  match f_s with
  | SProcess t ->
    let (σ, b, fs, input) = interp_process t Γ in
    (σ, b, {Γ with c_fs = fs; c_input = input})
  ...
```

Body – Mutual recursion

```
let rec interp_term (t: term) (Γ: context)
                    (stdout: ref string) : (bool, context)
=
  match t with
  | TAsString xs s ->
    let (σ, b, Γ') = interp_sexpr s Γ in
    let Γ'' = update_senv Γ' xs σ in
    if b then (true, Γ'') else raise (EFatal Γ'')
  ...

with interp_sfrag_aux (fs: sfrag) (Γ: context) (previous: bool)
                      : (string, bool, context)
=
  match fs with
  | SProcess t ->
    let (σ, b, fs, input) = interp_process t Γ in
    (σ, b, {Γ with c_fs = fs; c_input = input})
  ...
```

Soundness of the interpreter

We write $t_{/\Gamma} \mapsto \sigma \star b_{/\Gamma'}$ for: "*on the input consisting of* t, Γ *and a reference, the interpreter writes* σ *at the end of that reference and terminates:*

- *normally and outputs* (b, Γ')*;*
- *with an exception corresponding to the behaviour* b *that carries* Γ'*."*

Theorem (Soundness of the interpreter)

For all t, Γ, σ, b *and* Γ'*: if*

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

then

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

Soundness of the interpreter

We write $t_{/\Gamma} \mapsto \sigma \star b_{/\Gamma'}$ for: "*on the input consisting of t, Γ and a reference, the interpreter writes σ at the end of that reference and terminates:*

- *normally and outputs (b, Γ');*
- *with an exception corresponding to the behaviour b that carries Γ'.*"

Theorem (Soundness of the interpreter)

For all t, Γ, σ, b and Γ': if

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

then

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

Contract

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

  returns { (b, Γ') -> exists σ.
    !stdout = concat (old !stdout) σ
    /\ eval_term t Γ σ (BNormal b) Γ' }

  raises  { EFatal Γ' -> exists σ.
    !stdout = concat (old !stdout) σ
    /\ eval_term t Γ σ BFatal Γ' }

  raises  { EReturn (b, Γ') -> exists σ.
    !stdout = concat (old !stdout) σ
    /\ eval_term t Γ σ (BReturn b)  Γ' }

  raises  { EExit (b, Γ') -> exists σ.
    !stdout = concat (old !stdout) σ
    /\ eval_term t Γ σ (BExit b)  Γ' }
```

Contract

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

  returns { (b, Γ') -> exists σ.
    !stdout = concat (old !stdout) σ
    /\ eval_term t Γ σ (BNormal b) Γ' }

  raises  { EFatal Γ' -> exists σ.
    !stdout = concat (old !stdout) σ
    /\ eval_term t Γ σ BFatal Γ' }

  raises  { EReturn (b, Γ') -> exists σ.
    !stdout = concat (old !stdout) σ
    /\ eval_term t Γ σ (BReturn b)  Γ' }

  raises  { EExit (b, Γ') -> exists σ.
    !stdout = concat (old !stdout) σ
    /\ eval_term t Γ σ (BExit b)  Γ' }
```

Table of Contents

Why it is hard

- `stdout` is a reference:

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

- Usual fix: provide a witness as a ghost return value.
- Cannot work here because of exceptions: we would need to catch them all and all the time!

Never mind, there are provers based on superposition, let's use them.

Why it is hard

- `stdout` is a reference:

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

- Usual fix: provide a witness as a ghost return value.
- Cannot work here because of exceptions: we would need to catch them all and all the time!

Never mind, there are provers based on superposition, let's use them.

Why it is hard

- `stdout` is a reference:

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

- Usual fix: provide a witness as a ghost return value.
- Cannot work here because of exceptions: we would need to catch them all and all the time!

Never mind, there are provers based on superposition, let's use them.

Why it is hard

- `stdout` is a reference:

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

- Usual fix: provide a witness as a ghost return value.
- Cannot work here because of exceptions: we would need to catch them all and all the time!

Never mind, there are provers based on superposition, let's use them.

Why it is hard

- `stdout` is a reference:

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

- Usual fix: provide a witness as a ghost return value.
- Cannot work here because of exceptions: we would need to catch them all and all the time!

Never mind, there are provers based on superposition, let's use them.

But it works!

- 117 proof obligations;
- 190s on my machine;
- Uses Alt-Ergo, Z3 and E (crucial);
- No Coq proof.

But it works!

- 117 proof obligations;
- 190s on my machine;
- Uses Alt-Ergo, Z3 and E (crucial);
- No Coq proof.

But it works!

- 117 proof obligations;
- 190s on my machine;
- Uses Alt-Ergo, Z3 and E (crucial);
- No Coq proof.

Table of Contents

An other sound interpreter

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

  returns { (b, Γ') -> exists σ.
    !stdout = concat (old !stdout) σ
    /\ eval_term t Γ σ (BNormal b)  Γ' }

  ...
=
  while true do
    ()
  done
```

Table of Contents

Completeness of the interpreter

Theorem (Completeness of the interpreter)

For all t, Γ, σ, b *and* Γ'*: if*

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

then

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

On paper:

- We have the soundness,
- We can prove functionality of the predicate,
- Thanks to them, we can prove the termination,
- All of that gives us the completeness.

Completeness of the interpreter

Theorem (Completeness of the interpreter)

For all t, Γ, σ, b *and* Γ'*: if*

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

then

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

On paper:

- We have the soundness,
- We can prove functionality of the predicate,
- Thanks to them, we can prove the termination,
- All of that gives us the completeness.

Completeness of the interpreter

Theorem (Completeness of the interpreter)

For all t, Γ, σ, b *and* Γ': *if*

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

then

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

On paper:

- We have the soundness,
- We can prove functionality of the predicate,
- Thanks to them, we can prove the termination,
- All of that gives us the completeness.

Completeness of the interpreter

Theorem (Completeness of the interpreter)

For all t, Γ, σ, b *and* Γ'*: if*

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

then

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

On paper:

- We have the soundness,
- We can prove functionality of the predicate,
- Thanks to them, we can prove the termination,
- All of that gives us the completeness.

Completeness of the interpreter

Theorem (Completeness of the interpreter)

For all t, Γ, σ, b and Γ': if

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

then

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

On paper:

- We have the soundness,
- We can prove functionality of the predicate,
- Thanks to them, we can prove the termination,
- All of that gives us the completeness.

Completeness of the interpreter

Theorem (Completeness of the interpreter)

For all t, Γ, σ, b *and* Γ': *if*

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

then

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

On paper:

- We have the soundness,
- We can prove functionality of the predicate,
- Thanks to them, we can prove the termination,
- All of that gives us the completeness.

Completeness of the interpreter – In Why3?

Theorem (Completeness of the interpreter)

For all t, Γ, σ, b and Γ', if:

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

then

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

In Why3:

- We have the soundness, but we can't use it in the termination,
- We can prove the functionality,
- Thanks to it, and by re-proving the soundness on-the-fly, we can prove the termination,

Completeness of the interpreter – In Why3?

Theorem (Completeness of the interpreter)

For all t, Γ, σ, b *and* Γ', *if:*

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

then

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

In Why3:

- We have the soundness, but we can't use it in the termination,
- We can prove the functionality,
- Thanks to it, and by re-proving the soundness on-the-fly, we can prove the termination,

Completeness of the interpreter – In Why3?

Theorem (Completeness of the interpreter)

For all t, Γ, σ, b *and* Γ', *if:*

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

then

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

In Why3:

- We have the soundness, but we can't use it in the termination,
- We can prove the functionality,
- Thanks to it, and by re-proving the soundness on-the-fly, we can prove the termination,

Completeness of the interpreter – In Why3?

Theorem (Completeness of the interpreter)

For all t, Γ, σ, b *and* Γ', *if:*

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

then

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

In Why3:

- We have the soundness, but we can't use it in the termination,
- We can prove the functionality,
- Thanks to it, and by re-proving the soundness on-the-fly, we can prove the termination,

Functionality and termination

Theorem (Functionnality of the predicate)

For all t, Γ, σ_1, σ_2, b_1, b_2, Γ_1, Γ_2, *if:*

$$t_{/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad \textit{and} \qquad t_{/\Gamma} \Downarrow \sigma_2 \star b_{2/\Gamma_2}$$

then:

$$\sigma_1 = \sigma_2 \qquad \textit{and} \qquad b_1 = b_2 \qquad \textit{and} \qquad \Gamma_1 = \Gamma_2$$

Theorem (Termination of the interpreter)

For all t, Γ, σ, b, Γ', *if:*

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

then the interpreter terminates when given t, Γ *and a reference.*

Why we need the soundness and the functionality in the proof of termination

Case of the sequence (with non-exceptional behaviours):

```
| TSeq t1 t2 ->
  let (_, Γ1) = interp_term t1 Γ stdout in
  interp_term t2 Γ1 stdout
```

We know that:

$$\begin{aligned} \exists \sigma b \Gamma''. \quad & (t_1\ ;\ t_2)_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma''} \\ \wedge\ (\exists \sigma' b' \Gamma'. \quad & {t_1}_{/\Gamma} \Downarrow \sigma' \star b'_{/\Gamma'} \\ \wedge \quad & {t_2}_{/\Gamma'} \Downarrow \sigma \star b_{/\Gamma''} \\ \wedge \quad & b' \in \{\mathrm{True}, \mathrm{False}\}) \end{aligned}$$

But we need to say that that Γ' is in fact Γ_1.
Hence the need for the soundness and the functionality.

Why we need the soundness and the functionality in the proof of termination

Case of the sequence (with non-exceptional behaviours):

```
| TSeq t₁ t₂ ->
  let (_, Γ₁) = interp_term t₁ Γ stdout in
  interp_term t₂ Γ₁ stdout
```

We know that:

$$\begin{aligned} \exists \sigma b \Gamma''.\quad & (t_1\ ;\ t_2)_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma''} \\ \wedge\ (\exists \sigma' b' \Gamma'.\quad & t_{1/\Gamma} \Downarrow \sigma' \star b'_{/\Gamma'} \\ \wedge\quad & t_{2/\Gamma'} \Downarrow \sigma \star b_{/\Gamma''} \\ \wedge\quad & b' \in \{\mathrm{True}, \mathrm{False}\}) \end{aligned}$$

But we need to say that that Γ' is in fact Γ_1.

Hence the need for the soundness and the functionality.

Why we need the soundness and the functionality in the proof of termination

Case of the sequence (with non-exceptional behaviours):

```
| TSeq t₁ t₂ ->
  let (_, Γ₁) = interp_term t₁ Γ stdout in
  interp_term t₂ Γ₁ stdout
```

We know that:

$$\begin{aligned} \exists \sigma b \Gamma''. \quad & (t_1 \;;\; t_2)_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma''} \\ \wedge\, (\exists \sigma' b' \Gamma'. \quad & {t_1}_{/\Gamma} \Downarrow \sigma' \star b'_{/\Gamma'} \\ \wedge \quad & {t_2}_{/\Gamma'} \Downarrow \sigma \star b_{/\Gamma''} \\ \wedge \quad & b' \in \{\mathrm{True}, \mathrm{False}\}) \end{aligned}$$

But we need to say that that Γ' is in fact Γ_1.

Hence the need for the soundness and the functionality.

Why we need the soundness and the functionality in the proof of termination

Case of the sequence (with non-exceptional behaviours):

```
| TSeq t₁ t₂ ->
  let (_, Γ₁) = interp_term t₁ Γ stdout in
  interp_term t₂ Γ₁ stdout
```

We know that:

$$\begin{aligned} \exists \sigma b \Gamma''. \quad & (t_1 \ ; \ t_2)_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma''} \\ \wedge (\exists \sigma' b' \Gamma'. \quad & t_{1/\Gamma} \Downarrow \sigma' \star b'_{/\Gamma'} \\ \wedge \quad & t_{2/\Gamma'} \Downarrow \sigma \star b_{/\Gamma''} \\ \wedge \quad & b' \in \{\mathrm{True}, \mathrm{False}\}) \end{aligned}$$

But we need to say that that Γ' is in fact Γ_1.

Hence the need for the soundness and the functionality

Why we need the soundness and the functionality in the proof of termination

Case of the sequence (with non-exceptional behaviours):

```
| TSeq t₁ t₂ ->
  let (_, Γ₁) = interp_term t₁ Γ stdout in
  interp_term t₂ Γ₁ stdout
```

We know that:

$$\begin{aligned} \exists \sigma b \Gamma''. \quad & (t_1 \ ; \ t_2)_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma''} \\ \wedge (\exists \sigma' b' \Gamma'. \quad & {t_1}_{/\Gamma} \Downarrow \sigma' \star b'_{/\Gamma'} \\ \wedge \quad & {t_2}_{/\Gamma'} \Downarrow \sigma \star b_{/\Gamma''} \\ \wedge \quad & b' \in \{\text{True}, \text{False}\}) \end{aligned}$$

But we need to say that that Γ' is in fact Γ_1.
Hence the need for the soundness and the functionality.

What do we need, then?

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

  requires { exists σ b Γ'. eval_term t Γ σ b Γ' }

  variant { ... }

  returns { (b, Γ') -> exists σ.
    !stdout = concat (old !stdout) σ
    /\ eval_term t Γ σ (BNormal b) Γ' }

  ...
```

Now the question is: what variant are we going to use?

What do we need, then?

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

  requires { exists σ b Γ'. eval_term t Γ σ b Γ' }

  variant { ... }

  returns { (b, Γ') -> exists σ.
    !stdout = concat (old !stdout) σ
    /\ eval_term t Γ σ (BNormal b) Γ' }

  ...
```

Now the question is: what variant are we going to use?

Table of Contents

Let us find a variant

Terms are structurally decreasing? **Wrong.**

$$\frac{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 = \text{True} \\ t_{2/\Gamma} \Downarrow \sigma_2 \star b_{2/\Gamma_2} \qquad b_2 \in \{\text{True}, \text{False}\} \\ (\textbf{while } t_1 \textbf{ do } t_2)_{/\Gamma_2} \Downarrow \sigma_3 \star b_{3/\Gamma_3}}{(\textbf{while } t_1 \textbf{ do } t_2)_{/\Gamma} \Downarrow \sigma_1\sigma_2\sigma_3 \star b_{3/\Gamma_3}}$$

Proofs are structurally decreasing?
True, but we can't manipulate them in Why3.

Can we use the *height* or the *size* of the proof tree?

Let us find a variant

Terms are structurally decreasing? **Wrong.**

$$\frac{\begin{array}{c} t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 = \mathrm{True} \\ t_{2/\Gamma} \Downarrow \sigma_2 \star b_{2/\Gamma_2} \qquad b_2 \in \{\mathrm{True}, \mathrm{False}\} \\ (\textbf{while } t_1 \textbf{ do } t_2)_{/\Gamma_2} \Downarrow \sigma_3 \star b_{3/\Gamma_3} \end{array}}{(\textbf{while } t_1 \textbf{ do } t_2)_{/\Gamma} \Downarrow \sigma_1\sigma_2\sigma_3 \star b_{3/\Gamma_3}}$$

Proofs are structurally decreasing?
True, but we can't manipulate them in Why3.

Can we use the *height* or the *size* of the proof tree?

Let us find a variant

Terms are structurally decreasing? **Wrong.**

$$\frac{t_1{}_{/\Gamma} \Downarrow \sigma_1 \star b_1{}_{/\Gamma_1} \qquad b_1 = \mathsf{True} \\ t_2{}_{/\Gamma} \Downarrow \sigma_2 \star b_2{}_{/\Gamma_2} \qquad b_2 \in \{\mathsf{True}, \mathsf{False}\} \\ (\textbf{while } t_1 \textbf{ do } t_2)_{/\Gamma_2} \Downarrow \sigma_3 \star b_3{}_{/\Gamma_3}}{(\textbf{while } t_1 \textbf{ do } t_2)_{/\Gamma} \Downarrow \sigma_1\sigma_2\sigma_3 \star b_3{}_{/\Gamma_3}}$$

Proofs are structurally decreasing?

True, but we can't manipulate them in Why3.

Can we use the *height* or the *size* of the proof tree?

Let us find a variant

Terms are structurally decreasing? **Wrong.**

$$\frac{\begin{array}{c} {t_1}_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1} \qquad b_1 = \mathsf{True} \\ {t_2}_{/\Gamma} \Downarrow \sigma_2 \star {b_2}_{/\Gamma_2} \qquad b_2 \in \{\mathsf{True}, \mathsf{False}\} \\ (\textbf{while } t_1 \textbf{ do } t_2)_{/\Gamma_2} \Downarrow \sigma_3 \star {b_3}_{/\Gamma_3} \end{array}}{(\textbf{while } t_1 \textbf{ do } t_2)_{/\Gamma} \Downarrow \sigma_1\sigma_2\sigma_3 \star {b_3}_{/\Gamma_3}}$$

Proofs are structurally decreasing?
True, but we can't manipulate them in Why3.

Can we use the *height* or the *size* of the proof tree?

Let us find a variant

Terms are structurally decreasing? **Wrong.**

$$\frac{\begin{array}{cc} t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} & b_1 = \mathsf{True} \\ t_{2/\Gamma} \Downarrow \sigma_2 \star b_{2/\Gamma_2} & b_2 \in \{\mathsf{True}, \mathsf{False}\} \\ \multicolumn{2}{c}{(\textbf{while } t_1 \textbf{ do } t_2)_{/\Gamma_2} \Downarrow \sigma_3 \star b_{3/\Gamma_3}} \end{array}}{(\textbf{while } t_1 \textbf{ do } t_2)_{/\Gamma} \Downarrow \sigma_1\sigma_2\sigma_3 \star b_{3/\Gamma_3}}$$

Proofs are structurally decreasing?
True, but we can't manipulate them in Why3.

Can we use the *height* or the *size* of the proof tree?

Why it does not work – 1

Superposition provers are bad with arithmetic.

Patch: replace it with simple successor arithmetic.

But we would still need to talk about:

- addition and subtraction (for sizes);
- maximum and inequalities (for heights).

Why it does not work – 1

Superposition provers are bad with arithmetic.

Patch: replace it with simple successor arithmetic.

But we would still need to talk about:

- addition and subtraction (for sizes);
- maximum and inequalities (for heights).

Why it does not work – 1

Superposition provers are bad with arithmetic.

Patch: replace it with simple successor arithmetic.

But we would still need to talk about:

- addition and subtraction (for sizes);
- maximum and inequalities (for heights).

Why it does not work – 2

When we know the size of a proof, we cannot deduce from it the size of the proofs of the premises.

Patch: return the “unused” size.

But:

- Exceptions would have to carry that number too;
- We would have to catch all the exceptions to update that number.

Why it does not work – 2

When we know the size of a proof, we cannot deduce from it the size of the proofs of the premises.

Patch: return the “unused” size.

But:

- Exceptions would have to carry that number too;
- We would have to catch all the exceptions to update that number.

Why it does not work – 2

When we know the size of a proof, we cannot deduce from it the size of the proofs of the premises.

Patch: return the “unused” size.

But:

- Exceptions would have to carry that number too;
- We would have to catch all the exceptions to update that number.

Why it does not work – 3

We cannot deduce from the height of a proof the heights of the premises (only an upper bound)**.**

Patch: use inequalities in the pre- and post-conditions or in the predicate.

But it means more work:

- to define the pre- and post-conditions or the predicate;
- for the provers.

Why it does not work – 3

We cannot deduce from the height of a proof the heights of the premises (only an upper bound)**.**

Patch: use inequalities in the pre- and post-conditions or in the predicate.

But it means more work:

- to define the pre- and post-conditions or the predicate;
- for the provers.

Why it does not work – 3

We cannot deduce from the height of a proof the heights of the premises (only an upper bound)**.**

Patch: use inequalities in the pre- and post-conditions or in the predicate.

But it means more work:

- to define the pre- and post-conditions or the predicate;
- for the provers.

Table of Contents

Back to square one

We still want to say that proofs are structurally decreasing.

We add a `skeleton` type:

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

It represents the "shape" of the proof.

Back to square one

We still want to say that proofs are structurally decreasing.

We add a `skeleton` type:

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

It represents the “shape” of the proof.

Back to square one

We still want to say that proofs are structurally decreasing.

We add a `skeleton` type:

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

It represents the “shape” of the proof.

Put them everywhere – In the predicate

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

| EvalT_Seq_Normal : forall t₁ Γ σ₁ b₁ Γ₁ t₂ σ₂ b₂ Γ₂ sk1 sk2.
  eval_term t₁ Γ σ₁ (BNormal b₁) Γ₁ sk1 ->
  eval_term t₂ Γ₁ σ₂ b₂ Γ₂ sk2 ->
  eval_term (TSeq t₁ t₂) Γ (concat σ₁ σ₂) b₂ Γ₂ (S2 sk1 sk2)

| EvalT_Seq_Error : forall t₁ Γ σ₁ b₁ Γ₁ t₂ sk.
  eval_term t₁ Γ σ₁ b₁ Γ₁ sk ->
  (match b₁ with BNormal _ -> false | _ -> true end) ->
  eval_term (TSeq t₁ t₂) Γ σ₁ b₁ Γ₁ (S1 sk)
```

Put them everywhere – In the contract

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

  requires { exists s b g'. eval_term t g s b g' sk }

  variant { sk }

  returns { (b, Γ') -> exists σ.
    !stdout = concat (old !stdout) σ
    /\ eval_term t Γ σ (BNormal b)  Γ' sk }
```

Define some helpers

```
let ghost skeleton12 (sk: skeleton)

  requires { match sk with
    | S1 _ | S2 _ _ -> true
    | _ -> false
    end }

  ensures { match sk with
    | S1 sk1 | S2 sk1 _ -> result = sk1
    | _ -> false
    end }

  = match sk with
    | S1 sk1 | S2 sk1 _ -> sk1
    | _ -> absurd
    end
```

Define some helpers

```
let ghost skeleton12 (sk: skeleton)
  requires { match sk with S1 _ | S2 _ _ -> true | _ -> false
  ensures { match sk with S1 sk1 | S2 sk1 _ -> result = sk1 |
  = match sk with S1 sk1 | S2 sk1 _ -> sk1 | _ -> absurd end
```

The following:

```
let ghost sk1 = skeleton12 sk in
```

reads: *“We know that* `sk` *can only have one or two children and we name the first one* `sk1`*.”*

Define some helpers

```
let ghost skeleton12 (sk: skeleton)
  requires { match sk with S1 _ | S2 _ _ -> true | _ -> false
  ensures { match sk with S1 sk1 | S2 sk1 _ -> result = sk1 |
  = match sk with S1 sk1 | S2 sk1 _ -> sk1 | _ -> absurd end
```

The following:

```
let ghost sk1 = skeleton12 sk in
```

reads: *“We know that* `sk` *can only have one or two children and we name the first one* `sk1`.*”*

Put them everywhere – In the code

```
| TSeq t₁ t₂ ->
  let ghost sk1 = skeleton12 sk in
  let (_, Γ₁) = interp_term t₁ Γ stdout sk1 in
  let ghost (_, sk2) = skeleton2 sk in
  interp_term t₂ Γ₁ stdout sk2
```

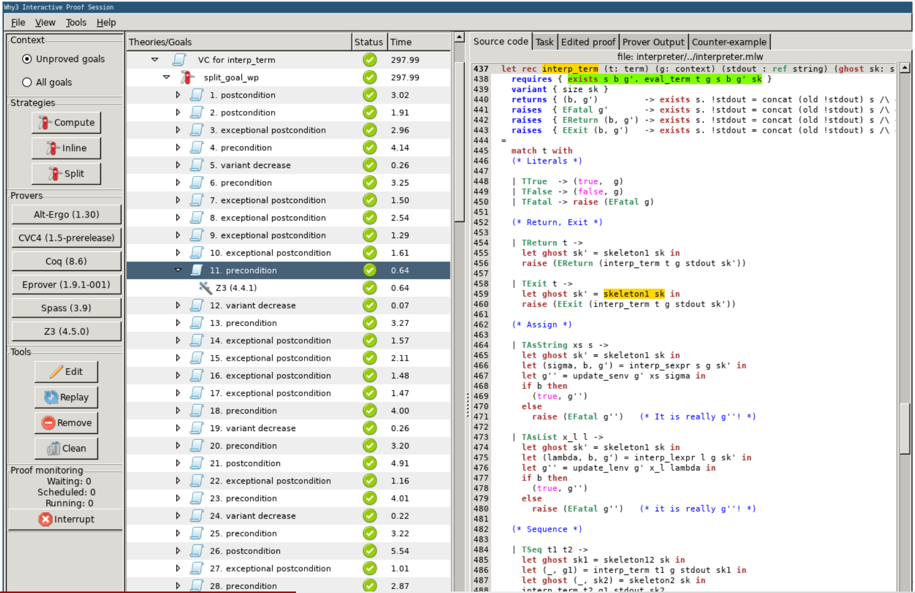
```
| TIf t₁ t₂ t₃ ->
  let (b₁, Γ₁) =
    try
      let ghost sk1 = skeleton12 sk in
      interp_term t₁ Γ stdout sk1
    with
      EFatal Γ' -> (false, Γ')
    end
  in
  let ghost (_, sk2) = skeleton2 sk in
  interp_term (if b₁ then t₂ else t₃) Γ₁ stdout sk2
```

Put them everywhere – In the code

```
| TSeq t₁ t₂ ->
  let ghost sk1 = skeleton12 sk in
  let (_, Γ₁) = interp_term t₁ Γ stdout sk1 in
  let ghost (_, sk2) = skeleton2 sk in
  interp_term t₂ Γ₁ stdout sk2
```

```
| TIf t₁ t₂ t₃ ->
  let (b₁, Γ₁) =
    try
      let ghost sk1 = skeleton12 sk in
      interp_term t₁ Γ stdout sk1
    with
      EFatal Γ' -> (false, Γ')
    end
  in
  let ghost (_, sk2) = skeleton2 sk in
  interp_term (if b₁ then t₂ else t₃) Γ₁ stdout sk2
```

And it's all green!



Theories/Goals	Status	Time
VC for interp_term	✓	297.99
split_goal_wp	✓	297.99
1. postcondition	✓	3.02
2. postcondition	✓	1.91
3. exceptional postcondition	✓	2.96
4. precondition	✓	4.14
5. variant decrease	✓	0.26
6. precondition	✓	3.25
7. exceptional postcondition	✓	1.50
8. exceptional postcondition	✓	2.54
9. exceptional postcondition	✓	1.29
10. exceptional postcondition	✓	1.61
11. precondition	✓	0.64
Z3 (4.4.1)	✓	0.64
12. variant decrease	✓	0.07
13. precondition	✓	3.27
14. exceptional postcondition	✓	1.57
15. exceptional postcondition	✓	2.11
16. exceptional postcondition	✓	1.48
17. exceptional postcondition	✓	1.47
18. precondition	✓	4.00
19. variant decrease	✓	0.26
20. precondition	✓	3.20
21. postcondition	✓	4.91
22. exceptional postcondition	✓	1.16
23. precondition	✓	4.01
24. variant decrease	✓	0.22
25. precondition	✓	3.22
26. postcondition	✓	5.54
27. exceptional postcondition	✓	1.01
28. precondition	✓	2.87

file: interpreter/../interpreter.mlw

```
let rec interp_term (t: term) (g: context) (stdout : ref string) (ghost sk: s
  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 /\
  raises  { EFatal g'        -> exists s. !stdout = concat (old !stdout) s /\
  raises  { EReturn (b, g') -> exists s. !stdout = concat (old !stdout) s /\
  raises  { EExit (b, g')   -> exists s. !stdout = concat (old !stdout) s /\
=
  match t with
  (* Literals *)

  | TTrue  -> (true,  g)
  | TFalse -> (false, g)
  | TFatal -> raise (EFatal g)

  (* Return, Exit *)

  | TReturn t ->
    let ghost sk' = skeleton1 sk in
    raise (EReturn (interp_term t g stdout sk'))

  | TExit t ->
    let ghost sk' = skeleton1 sk in
    raise (EExit (interp_term t g stdout sk'))

  (* Assign *)

  | TAsString xs s ->
    let ghost sk' = skeleton1 sk in
    let (sigma, b, g') = interp_sexpr s g sk' in
    let g'' = update_senv g' xs sigma in
    if b then
      (true, g'')
    else
      raise (EFatal g'')   (* It is really g''! *)

  | TAsList x_l l ->
    let ghost sk' = skeleton1 sk in
    let (lambda, b, g') = interp_lexpr l g sk' in
    let g'' = update_lenv g' x_l lambda in
    if b then
      (true, g'')
    else
      raise (EFatal g'')   (* it is really g''! *)

  (* Sequence *)

  | TSeq t1 t2 ->
    let ghost sk1 = skeleton12 sk in
    let (_, g1) = interp_term t1 g stdout sk1 in
    let ghost (_, sk2) = skeleton2 sk in
    interp_term t2 g1 stdout sk2
```

And it's all green!

- 233 proof obligations;
- 510s on my machine;
- Uses Alt-Ergo, Z3 and E;
- Still no Coq proof.

And it's all green!

- 233 proof obligations;
- 510s on my machine;
- Uses Alt-Ergo, Z3 and E;
- Still no Coq proof.

And it's all green!

- 233 proof obligations;
- 510s on my machine;
- Uses Alt-Ergo, Z3 and E;
- Still no Coq proof.

Other things about skeletons

- Generalisable, if we want more than the shape;
- Help in writing recursion in case of mutually recursive types (because there is now a common structurally decreasing value);
- Can really be added automatically to inductive predicates;
- Works because:
    - the order of the premises is the order of the execution,
    - the proof tree looks pretty much like the recursive calls tree.

Other things about skeletons

- Generalisable, if we want more than the shape;
- Help in writing recursion in case of mutually recursive types (because there is now a common structurally decreasing value);
- Can really be added automatically to inductive predicates;
- Works because:
  - the order of the premises is the order of the execution,
  - the proof tree looks pretty much like the recursive calls tree.

Other things about skeletons

- Generalisable, if we want more than the shape;
- Help in writing recursion in case of mutually recursive types (because there is now a common structurally decreasing value);
- Can really be added automatically to inductive predicates;
- Works because:
  - the order of the premises is the order of the execution,
  - the proof tree looks pretty much like the recursive calls tree.

Other things about skeletons

- Generalisable, if we want more than the shape;
- Help in writing recursion in case of mutually recursive types (because there is now a common structurally decreasing value);
- Can really be added automatically to inductive predicates;
- Works because:
  - the order of the premises is the order of the execution,
  - the proof tree looks pretty much like the recursive calls tree.

Other things about skeletons

- Generalisable, if we want more than the shape;
- Help in writing recursion in case of mutually recursive types (because there is now a common structurally decreasing value);
- Can really be added automatically to inductive predicates;
- Works because:
    - the order of the premises is the order of the execution,
    - the proof tree looks pretty much like the recursive calls tree.

Thank you for your attention!

Questions? Comments? Suggestions?

Claude Marché, Nicolas Jeannerod and Ralf Treinen
A Formally Verified Interpreter for a Shell-like Programming Language
VSTTE, July 2017